Efficient Implementation of Image Compression-Postprocessing Algorithm Using a Digital Signal Processor

Nadir Sinaceur
Portland State University

THESIS APPROVAL

The abstract and thesis of Nadir Sinaceur for the Master of Science in Electrical and Computer Engineering were presented December 11, 1997, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]
Fu Li, Chair

[redacted]
Branimir Pejcinovic

[redacted]
Bradford Crain
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]
Lee W. Casperson
Department of Electrical Engineering

ABSTRACT

An abstract of the thesis of Nadir Sinaceur for the Master of Science in Electrical and Computer Engineering presented December 11, 1997.

Title: Efficient Implementation of Image Compression-Postprocessing Algorithm Using a Digital Signal Processor.

In this thesis, an attempt has been made to develop a fast way to implement a post-processing algorithm for image compression. All the previous tests for this postprocessing algorithm, which we will present, have been only software based and did not consider the time parameter.

For this purpose a new algorithm is used to compute the 2-D DCT transform. This change made the process a lot faster on a Sparc 5 workstation. We have then decided to further increase the speed of the post-processing scheme by implementimg it on the ADSP21020 chip.

The resuts show that such a chip can achieve a speed increase and that if the code is optimized a faster processing is even reachable.

EFFICIENT IMPLEMENTATION OF IMAGE COMPRESSION-POSTPROCESSING ALGORITHM USING A DIGITAL SIGNAL PROCESSOR

by

NADIR SINACEUR

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1998

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rise of the Internet and other new technologies has pushed the research on image compression. Indeed the need for transferring still images and video over a low bandwidth network at a very high speed has been growing exponentially in these last years. To get an idea of what the challenge is, we will consider a typical low resolution, color video image of 512 x 512 pixels. It requires approximately 6 x 10^6 bits. This means that for sending a small video file it would take an incredible amount of memory.

Luckily most of the data in this image is not equally important. Statistical redundancy as well as irrelevancy in regard to the eye of an observer makes the technique of image compression a very attractive solution for our purpose. It is well known that the human visual system shows sensitivity variations depending on the orientation, the light level and other signals. This property makes a lot of data in an image irrelevant to the human eye. As far as the redundancy goes, it can be either spectral between color planes or spectral bands, temporal between neighboring frames in a video sequence or spatial between neighboring pixels.

The study of image compression is based on those two parameters . An optimal image compression would be removing all the "unnecessary" data and keeping only the important ones that would make a difference to the observer if they weren't present. Two ways have been actively pursuit to achieve this goal. The first one, which is characterized as the lossless one, makes no compromise as far as the quality of the image received at the other end. In doing that, no data is ever lost but the compression ratio stays low around 2/1 to 4/1. The big advantage of this process is that it is a 100 % reversible.

On the other hand, the lossy compression uses all the "deficiencies" of the human eyes as well as the redundancies to eliminate all the non-essential pixels. In doing that it can achieve a very high compression ratio at the expense of a lower quality image. This second approach is currently widely used over low bandwidth networks. An example of a good application for this technique is the Internet videophone. Since the image of the caller doesn't have to be perfect in order for the receiver to understand what the caller means by any movement of his face.

Nevertheless, it is important that the artifacts caused by such loss of information in the image are kept to a minimum. This brings us to the two main artifact removal methods used nowadays in image processing. The first one or *pre-processing* is done at the encoder level using block overlap, different coding for edge blocks to remove the "ringing" effect, or DC calibration. The second one or *post processing* is done at the decoder level using for example a low pass filter to smoothen the block boundaries or edge adaptive filtering.

Today one of the most common standard for image compression scheme is the JPEG (Joint Photographic Experts Group) one. Since it is so widely spread it seemed normal to use it as a base for this work. We will therefore first explain in details the JPEG algorithm for lossy image compression. After that we will present the post-processing method from a theoretical point of view. This will reduce the blocking effect and ringing effects. Having done this we will then look at a more practical view of image compression by implementing the algorithm presented. In a first step we will do that on a workstation and after that we will simulate the same code on an Analog Devices ADSP21020 chip and compare the execution times. As a last step, we will optimize the code in order to get an efficient implementation of this algorithm.

Chapter 2

2 The JPEG Standard

2.1 Introduction

Image compression is the art and science of reducing the number of bits required to describe an image. This is done usually at the source or encoder level . The compressed data is then stored or transferred to the receiver or decoder in order to be reconstructed. This process has been standardized [1] so that applications could be created and hardware designed and optimized for such a standard.

2.2 Image Compression Models

There are many compression models depending on the way the image is coded. Two main classes are the predictive coding and the transform coding.

In the predictive coding class, the information is used to predict the new values and the difference is coded. An example of this class is given by the DPCM or "differential pulse code modulation". Given a beginning value, DPCM uses the coded differences between each sampled value to reconstruct the whole picture. If there is a strong correlation between samples, in other words if the differences between neighboring pixels are small, the DPCM method is a very reliable method.

In the transform coding, an image is represented by a discrete set of basis arrays called basis images, just as a one-dimensional signal can be represented by an orthogonal series of basis functions. In other words the pixels are transformed to another domain, in our case the frequency domain. Such a unitary transformation not only preserves the signal energy but also packs a large fraction of the average energy of the image into relatively few components of the transform coefficients. As a result of this, many of the transform coefficients contain little energy, which makes it easier to discard them if necessary. In any practical system, the compression process is followed at some point by decompression. Thus the transformation process has to be reversible so that X can be reconstructed from Y. It follows that, for a specific T there should be a U such that $X = UYU^t$.

The Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), Discrete Sine Transform as well as other transforms as the Karhunen-Loève Transform (KLT) are all unitary transforms, that fulfill this condition and basically would be suitable. However, the KLT has several implementation-related deficiencies, including the fact that the basis functions are image dependent. The other basis functions are image independent. The JPEG group chose the DCT transform after a selection process on a first blind assessment of subjective picture quality and on a second more rigorous selection. It is therefore one of the best if not the best transform for lossy image compression.

2.3 Encoder

2.3.1 DCT-Based Coding

For image processing applications, the forward 2-D DCT of an n x n block of pixels is expressed as

$$F(u,v) = \frac{4C(u)C(v)}{n^2} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} f(j,k) \cos\left[\frac{(2j+1)u\pi}{2n}\right] \cos\left[\frac{(2k+1)v\pi}{2n}\right] \qquad (2.1)$$

And the inverse 2-D DCT is defined as

$$f(j,k) = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} C(u)C(v)F(u,v) \cos\left[\frac{(2j+1)u\pi}{2n}\right] \cos\left[\frac{(2k+1)v\pi}{2n}\right] \qquad (2.2)$$

Where

$$C(w) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for} \quad w = 0 \\ 1 & \text{for} \quad w = 1,2,\ldots,n-1 \end{cases}$$

The DCT transformation decomposes each input block into a series of waveforms, each with a particular spacial frequency.

An important property of the 2-D DCT and IDCT transform is separability. We will see that from an implementation viewpoint, a row-column approach will simplify the hardware requirements. For now we will remember three main points that make this transformation very attractive.

1.The DCT basis is image independent and causes most of the energy to be concentrated in the upper left corner of the transformed matrix, as shown below.

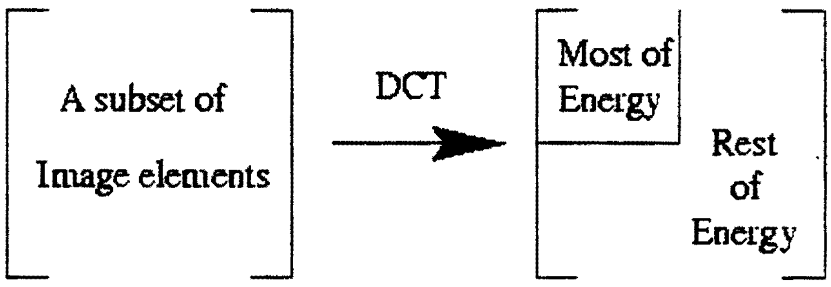

Figure 2.1. DCT Transformation

2.Since each coefficient obtained in frequency domain is the contribution of its correspondent waveform, the characteristics of the human visual system could be easily incorporated by modifying those values.

3.The DCT computations as expressed above, can be performed with fast algorithms that require fewer operations than the computations performed directly from these equations.

2.3.2 Quantization

In this lossy compression mode, some of the coefficients or weights will be deleted and therefore the corresponding waveforms will not be used during decompression. This process is referred to as the quantization. Mathematically this process is defined through a scaling of the coefficients using a user specified quantization table that is fixed for all blocks, followed by a rounding off to the nearest integer:

$$F^{*}(u,v) = round\ \left(\frac{F(u,v)}{Q(u,v)}\right) \qquad (2.4)$$

F*(u,v) and Q(u,v) represent the quantized coefficient and normalization matrix element. Each component of the quantization table is an 8-bit integer that determines the quantization step size and therefore the quality of the encoded image. A typical quantization table that is also used by the JPEG standard is:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Table 2.1. Luminance quantization table

Considering the original pixel matrix 2.5.

$$\begin{bmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 156 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 162 & 162 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 162 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{bmatrix} \qquad (2.5)$$

Then the quantized DCT output using Table 2.1 is given by 2.6.

$$\begin{bmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

The process of quantization has resulted in the zeroing out of many of the DCT coefficients. The specific design of Q depends on psychovisual characteristics and compression ratio considerations. The last step in this part of the encoder is a classification of the DC coefficients using the following DPCM model

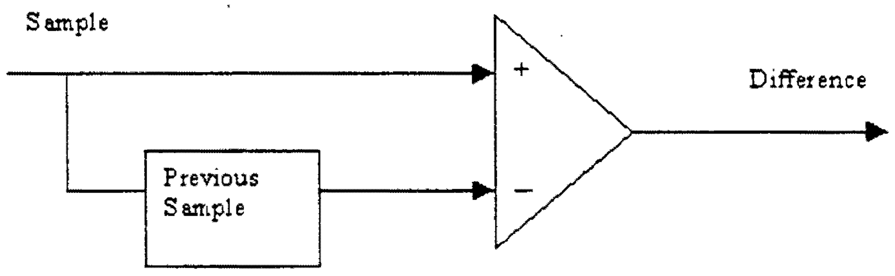

Figure 2.2. DPCM model

Due to the high degree of correlation of DC values among adjacent blocks, this model is very effective. The other 63 AC coefficients are zigzag scanned.

2.3.3 Zigzag Scan

Since most of the DCT coefficients have a value equal to 0 after quantization, JPEG uses a zigzag pattern to convert the 2-D coefficients into a 1-D sequence. This results in the coefficients being approximately arranged in decreasing order of their average energy. This creates large runs of zero values that suit well the following entropy model. The pattern followed starts from the left upper corner to the right lower corner along the diagonal paths shown below. (Ordered from 0 to 63):

$$\begin{bmatrix} 0 & 1 & 5 & 6 & 14 & 15 & 27 & 28 \\ 2 & 4 & 7 & 13 & 16 & 26 & 29 & 42 \\ 3 & 8 & 12 & 17 & 25 & 30 & 41 & 48 \\ 9 & 11 & 18 & 24 & 31 & 40 & 47 & 53 \\ 10 & 19 & 23 & 32 & 39 & 46 & 52 & 57 \\ 20 & 22 & 33 & 38 & 45 & 51 & 56 & 60 \\ 21 & 34 & 37 & 44 & 50 & 55 & 59 & 62 \\ 35 & 36 & 43 & 49 & 54 & 58 & 61 & 63 \end{bmatrix}$$

Figure 2.3. Zigzag scan

In our case the coefficients are 15 0 -2 -1 -1 -1 0 0 -1 all Zeros. The ordered coefficients can now be efficiently represented using a combination of a run-length coding and Huffman coding scheme.

2.3.4 Entropy Coding

The last processing block in JPEG is the entropy coder. It improves overall performance by performing lossless coding on the quantized DCT coefficients. For this purpose two statistical models are used. One corresponds to the coding of the DC differences generated in the step before. The other is the coding of the AC coefficients.

2.3.5 Huffman Coding of the DC coefficients

The difference between the DC values of the two consecutive blocks is first categorized using a table lookup. This table shown in Table 2.2 consists of 12 categories, where the ith category contains all the differentials that can be represented by i bits. To each category k corresponds a set of Huffman codes (base codes) with a maximum codeword length of 16 bits.

Category	Coefficient Range
0	0
1	-1,1
2	-3,-2,2,3
3	-7,...,-4,4,...,7
4	-15,...,-8,8,...,15
5	-31,...,-16,16,...,31
6	-63,...,-32,32,...,63
7	-127,...,-64,64,...,127
8	-255,...,-128,128,...,255
9	-511,...,-256,256,...,511
10	-1023,...,-512,512,...,1023
11	-2047,...,-1024,1024,...,2047

Table2.2 DC and AC coefficient grouping

These codes do not completely describe the difference. Therefore, an additional k bits are sent to completely specify the sign and magnitude of a difference value in that category.

Category	Base Code	Length	Category	Base Code	Length
0	010	3	6	1110	10
1	011	4	7	11110	12
2	100	5	8	111110	14
3	00	5	9	1111110	16
4	101	7	A	11111110	18
5	110	8	B	111111110	20

Table 2.3. JPEG DC code

For example, if the DC value of the previous block is 24, then the difference $DC_i - DC_{i-1}$ = 15-24=-9. This number lies in category 4 of table 2.2, which gives us a base code of 101.

2.3.6 Huffman Coding of the AC Coefficients

As in the step before each AC coefficient can be described by the pair (size, amplitude). Since after the quantization most of the AC coefficients are zero, only the nonzero AC coefficients need to be coded. A run length coder yields the value of the next nonzero AC and a run, that is the number of zero AC coefficients preceding this one. This means that each nonzero AC coefficient can be described by the pair (run/size, amplitude). The value of run/size is Huffman coded, and the value of the amplitude (computed as in the case of the DC differentials) is appended to that code.

There are two special cases in the coding of AC coefficients, as follows: (1) The run-length value may be larger than 15. In that case JPEG uses the symbol (15/0) to denote a run-length of 15 zeros followed by a zero. If the runlength exceeds 16 zero coefficients, it is coded by using multiple symbols. In addition, if after a nonzero AC value all the remaining coefficients are zero, then the special symbol (0/0) denotes the end of block (EOB).

For the AC values, using the values in the table for luminance AC coefficients we obtain the following. The first non zero value .is (-2) and coded as 11100101. Following the same method we obtain for our example.

1010110/11100101/000/000/000/110110/1010

Zero Run	Category	Codelength	Codeword
0	1	2	00
0	2	2	01
0	3	3	100
0	4	4	1011
0	5	5	11010
0	6	6	111000
0	7	7	1111000
.	.	.	.
1	1	4	1100
1	2	6	111001
1	3	7	1111001
1	4	9	111110110
.	.	.	.
2	1	5	11011
2	2	8	11111000
.	.	.	.
3	1	6	111010
3	2	9	111110111
.	.	.	.

Table 2.4. JPEG AC Code

Zero Run	Category	Codelength	Codeword
4	1	6	111011
5	1	7	1111010
6	1	7	1111011
7	1	8	11111001
8	1	8	11111010
9	1	9	111111000
10	1	9	111111001
11	1	9	111111010
.	.	.	.
End of Block (EOB)		4	1010

Table 2.4 continued

2.4 Decoding

At the receiver side we perform entropy decoding by simply using the Huffman table used in the transmitter. This step corresponds to a simple lookup and results in the exact same coefficients obtained after quantization in the transmitter:

$$\begin{bmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.7}$$

We then use the same quantization table used in thequantization step of the encoder to decode the resulting matrix. This practically means multiplying the 8x8 block by the quantization matrix obtaining the following block:

$$\begin{bmatrix} 240 & 0 & -10 & 0 & 0 & 0 & 0 & 0 \\ -24 & -12 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & -13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.8}$$

This new block is then inversely transformed using the inverse DCT. The value of 128 is added back to obtain the end 8x8-pixel block

$$\begin{bmatrix} 144 & 146 & 149 & 152 & 154 & 156 & 156 & 156 \\ 148 & 150 & 152 & 154 & 156 & 156 & 156 & 156 \\ 155 & 156 & 157 & 158 & 158 & 157 & 156 & 155 \\ 160 & 161 & 161 & 162 & 161 & 159 & 157 & 155 \\ 163 & 163 & 164 & 163 & 162 & 160 & 158 & 156 \\ 163 & 163 & 164 & 164 & 162 & 160 & 158 & 157 \\ 160 & 161 & 162 & 162 & 162 & 161 & 159 & 158 \\ 158 & 159 & 161 & 161 & 162 & 161 & 159 & 158 \end{bmatrix} \qquad (2.9)$$

As we can see, the end block of pixels obtained at the receiver side is different then the original one. This is the reason why this compression scheme is called "lossy". In this example the errors range from –5 to 5.

One way to measure the "lossyness" or amount of information lost during this compression-decompression process is given by the (RMSE) (root mean square error)

$$RMSE = \sqrt{\frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left[o(i,j)-r(i,j)\right]^2} \qquad (2.10)$$

Where, N and M are width and height of the image in pixels, o is the original image and r is the reconstructed image.

In our case we obtain:

$$RMSE = \sqrt{\frac{1}{64}\sum_{i=1}^{7}\sum_{j=1}^{7}\left[o(i,j)-r(i,j)\right]^2} = 2.26 \tag{2.11}$$

Chapter 3

Blocking, Ringing & Post-processing

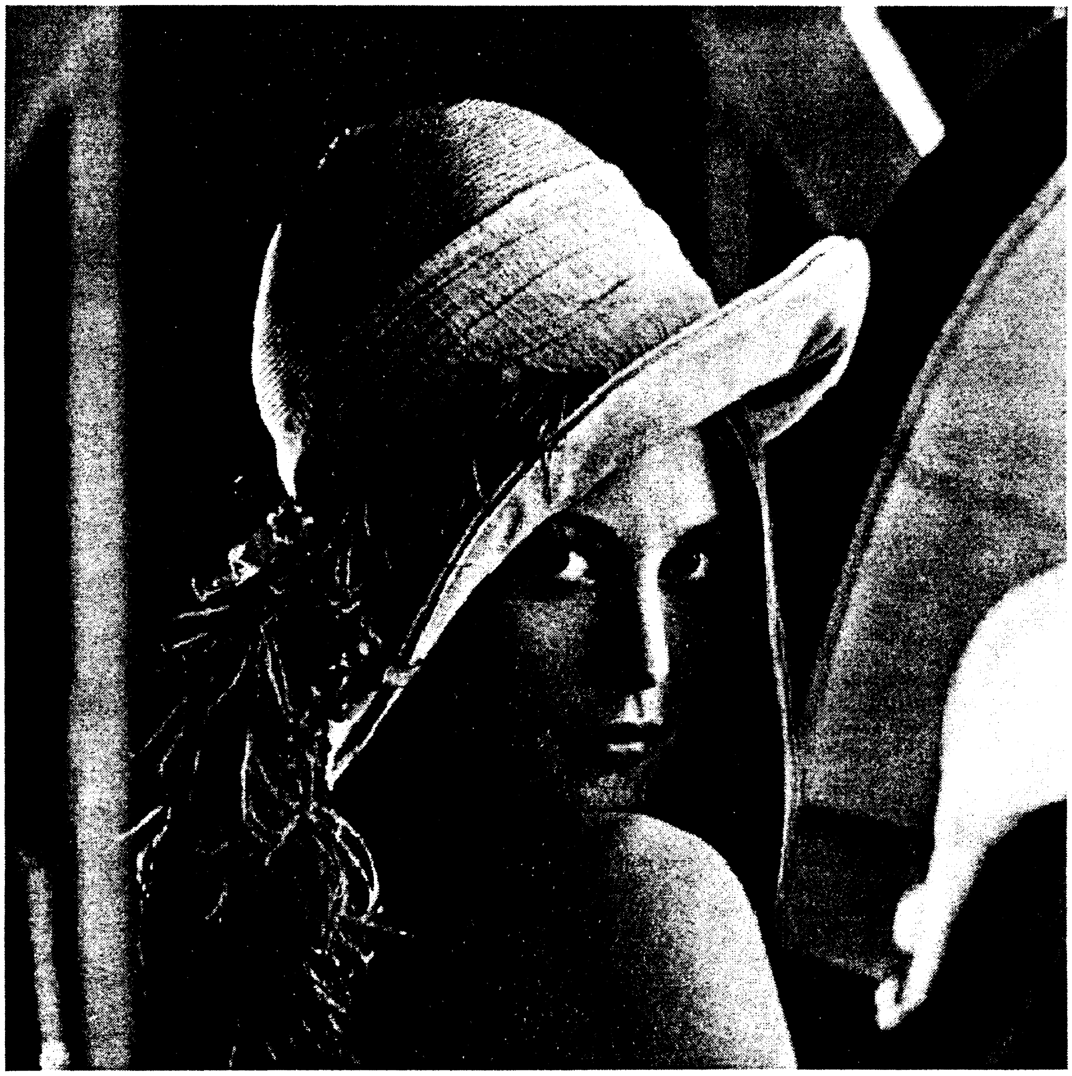

Figure 3.1:Original Lena picture 512x512

3.1 Introduction

In this chapter the artifacts created by the lossy compression are explained and several methods of artifact removal are presented.

3.2 Artifacts observed in JPEG

The two major artifacts, we observe after a lossy DCT based compression extended to low bit rates, are called the blocking effect and the ringing effect. The first artifact occurs when the DCT coefficient quantization step size is above the threshold for visibility. In that case, we clearly see discontinuities in grayscale values at the boundaries between blocks. The ringing artifact is the result of a coarse quantization that discards the high frequency DCT coefficients and causes contouring along sharp edges on the uniform background.

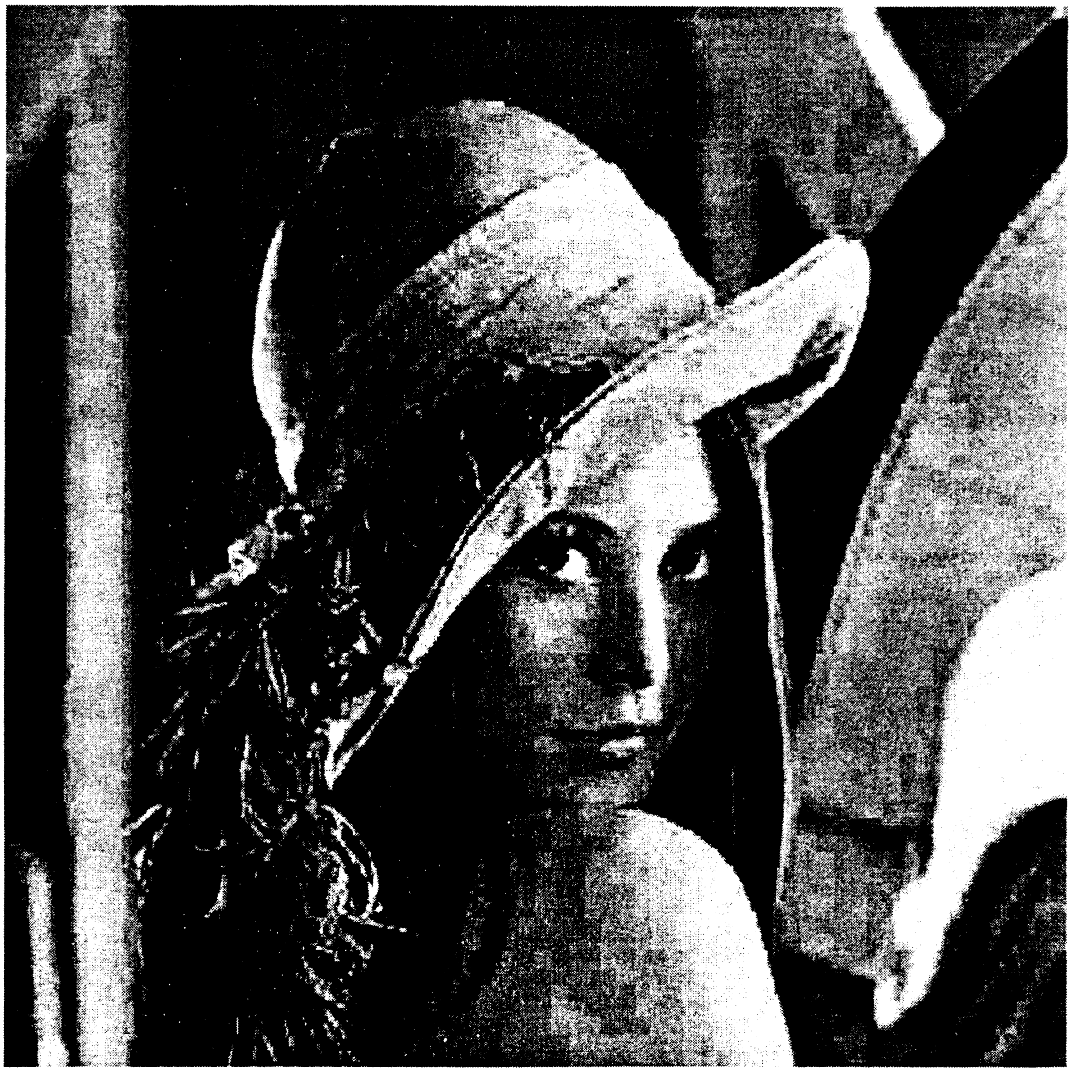
Figure 3.2:Lena picture encoded at 0.25 bits/pixel

3.3 Ways to reduce the compression artifacts

Basically two methods were used to achieve this reduction. Either at the encoder side using a different coding scheme or at the decoder side by using post-processing.

3.3.1 New Encoding Schemes

There are several ways to avoid the blocking artifacts. One of them is by dividing the image into overlapping blocks as reported in [2]. Another method *by Lynch et al* [3], is the Edge Compensated Transform Coding (ECTC), which preprocesses the image before the transform coding. At the encoder side as well as at the decoder side the blocks with ringing artifacts are detected and coded differently then the others. A method proposed by *Luo et al.*[4] tries to remove the blocking effect by calibrating the DC component of the blocks and not only focusing on the block boundary area. All these methods achieve a certain enhancement on the final decoded image. The main disadvantage they all have is the fact that they differ from the JPEG standard and are not standardized therefore very unlikely to be endorsed by the multimedia industry. On the other hand some post-processing method can achieve substantial enhancements while staying JPEG compliant.

3.3.2 Post-processing Schemes

These schemes are more attractive since they allow an improvement in the image while not requiring any changes to the encoded bit stream. *Reeve and Lim* [2] proposed a method based on low pass filtering the pixels that are adjacent to the block boundaries. They chose in this case a 3x3 Gaussian low pass filter as shown in Fig 3.3.

.0751	.1239	.0751
●	●	●
.1239	.2042	.1239
●	●	●
.0751	.1239	.0751
●	●	●

Figure 3.3: 3x3 Gaussian Low Pass Filter

This method can remove most of the blockiness but it is at the expense of creating a blurring effect.

Lynch et al. in [5] proposed an algorithm that applies a space-varying filter in low frequency blocks and edge blocks. First the edge blocks are detected in the spacial domain then the low frequency blocks are detected in the transform coefficient domain. The next step is to apply a low pass filter that has a size between 3x3 to 9x9, depending on the size of the flat region in the edge blocks. This algorithm showed it could improve the SNR with 0.4 dB at a low bit rate.

We will now present the algorithm chosen in this work, because of a higher improvement in SNR at a low bit rate as well as a subjective image improvement for the human eye.

3.4 The chosen Post-processing Algorithm

This algorithm is divided in three main categories, which are blocking artifact removal, ringing artifact removal, and fidelity constraint. The first step is a block classification by detecting the blocking artifacts and the ringing artifacts in the transform domain. Since the blocking artifacts are more visible in the low frequency blocks and the ringing artifacts appear along the sharp edges or high frequency blocks, we classify the low frequency and high frequency blocks in the transform domain. A block is declared as low frequency block if:

$$C_{DCT}(i,j) * K_{low} = \hat{0} \qquad (3.1)$$

It is marked as high frequency if:

$$C_{DCT}(i,j) * K_{high} \neq \hat{0} \qquad (3.2)$$

$C_{DCT}(i,j)$ is the 8x8 block containing the quantized DCT coefficients of block (i,j). * corresponds to an element-by-element multiplication. K_{low} and K_{high} are test matrices for detection of low frequency blocks and high frequency blocks, respectively. Ô is the 8x8 null matrix.

After a series of experiments K_{low} is chosen as:

$$K_{low} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad (3.3)$$

And K_{high} as:

$$K_{high} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad (3.4)$$

Having made this choice, it is now possible to locate the blocks with blocking effects and the ones with ringing effects in the JPEG encoded pictures at a low bit rate. Now that we have classified the different blocks we will apply different techniques to reduce those artifacts.

3.4.1 Removal of Blocking Artifacts

To reduce the blocking artifacts we have to consider the fact that all the pixels in those blocks have to be altered and not only the boundary ones. A good way of performing this is by applying a midpoint displacement interpolation to specific regions. Those regions are specified as the ones where four adjacent blocks are categorized as blocking blocks.

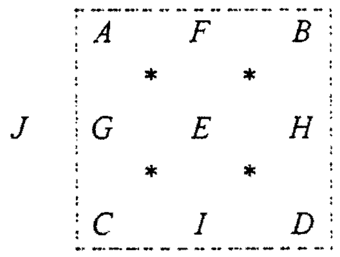

Figure 3.4:The Mid-point Displacement Interpolation

To apply this interpolation on one region, we choose a center pixel (4,4) from each of the four 8x8 adjacent blocks. These pixels are noted as A, B, C and D in Fig 3.4. The value of the pixel at location E, which is at equal distance from A, B, C and D is interpolated by taking the average values of these surrounding pixels. Recursively, the following pixels F, G, H and I are filled. Those pixels have two choices of values. Let us take for example the case of G. It is the center of pixels A, E, C and J. We have to determine now if J is part of a blocking block or not. If it is the case then G will take the value of the average of A, E, C, and J. If not it will keep its original value. The same way we can interpolate the values of F, H and I. The next stage is calculating the values of the pixels at location "*" using the same process. When all the pixels that can be interpolated have been altered most of the blocking artifacts will be reduced.

3.4.2 Removal of Ringing Artifact

In order to take care of the ringing artifact we need to detect the real edges corresponding to the ringing blocks. Indeed, it is important to conserve the edges whithin a block and apply an edge-adaptive lowpass filter on the smooth areas within this same block.

Edges can be thought of as pixel values of abrupt gray-level change. One gradient operator that extracts the edges is the Sobel operator. In the Sobel edge detection, a pixel at the location (i,j) is an edge pixel if g(i,j) > t . t being a certain threshold and g(i,j) being a bidirectional gradient. In the "Lena" picture case we have used the value 15 for t to detect the correct edges.

The edge pixels are marked with a black dot. We then scan each block pixel by pixel assigning each pixel to a certain region according to the neighboring pixels. by doing so we obtain the following example of a ringing block

a	●	*b*	*b*	●	*c*	*c*	*c*
a	*a*	●	*b*	●	*c*	*c*	*c*
a	*a*	*a*	●	●	*c*	*c*	●
a	*a*	●	●	●	●	●	*e*
●	●	*d*	*d*	*d*	●	*e*	*e*
d	*d*	*d*	*d*	*d*	●	*e*	*e*
d	*d*	*d*	*d*	●	*e*	*e*	*e*
d	*d*	*d*	●	*e*	*e*	*e*	*e*

Figure 3.5:Ringing block example

The next step is to smoothen all the areas that are separated by the edge pixels. We simply apply a low pass filter, which averages the values in each region individually. One problem that might occur while doing that is oversmoothing some regions. Therefore we apply the next explained fidelity constraint.

3.4.3 Fidelity Constraint

Now that we have taken care of the blocking and ringing artifacts in the spatial domain, it is time to make sure that our new DCT coefficients are within the allowed range by the quantization theory. For example, if the quantizer step size is 16 and the quantized DCT value is 3, then the unquantized DCT value must be in [40,55]. Therefore it is important to make sure that the new DCT coefficients are within received values plus or minus quantization bin value (half of the quantizer step size). If it is the case, these new DCT values will be used. If not , it will be replaced by the maximum or minimum value allowed in that range. In our example if the recovered DCT coefficient is larger than 40 and less then 55 , it will be used. But if it is larger then 55 or lower then 40, 55 or 40 will be chosen appropriately.

3.5 Image improvements

By using the algorithm we have described, we can achieve at least 0.6-dB improvement of peak signal-to-noise ratio (PSNR) over the standard JPEG compressed image at 0.25 bits/pixel.

$$PSNR = 20\log_{10}\frac{255}{RMSE} \qquad (3.5)$$

Where root mean-squared error (RMSE) is defined as

$$RMSE = \sqrt{\frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left[o(i,j)-r(i,j)\right]^2} \tag{3.5}$$

N and M are the width and height, respectively of the images in pixels, o is the original image and r the reconstructed image.

It is also worth noticing by looking at the recovered Lena image after post-processing that the subjective performance based on the human eye has been also improved. We will now discuss the issue of implementing this algorithm as software and hardware.

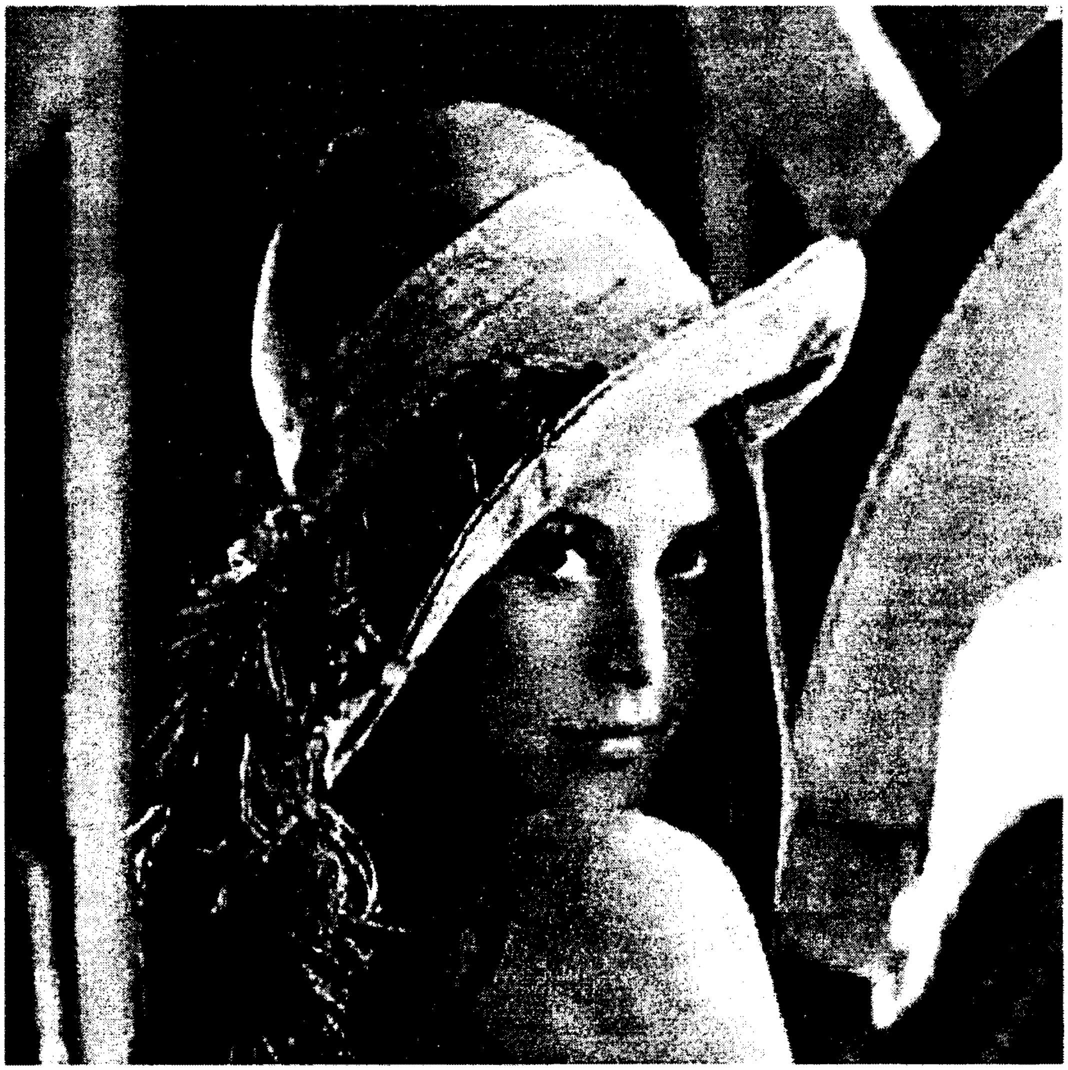

Figure 3.6:Lena after post-processing encoded at 0.25 bits/pixel

Chapter 4

Digital Signal Procesing Power

4.1 Introduction

The design of any image processing system must evaluate the processing power required to perform various application algorithms. Image processing systems are advancing in step with rapid technological development on a variety of fronts. Digital CCD cameras are providing high definition data, semiconductor memories are capable of storing high definition video data for processing and display, and video monitors are capable of displaying high quality images. With the explosion of the Internet, 3D modeling and multimedia the processing power requirements are substantial. For general-purpose desktop processing a top end PC will cope with only trivial applications making the use of hardware accelerators necessary for any serious work.

To this end Digital Signal Processors can be used to supply the necessary processing power. Rapid progress in semiconductor technology and processor architectures has enabled the use of programmable DSP's for many image-processing requirements. These requirements are application dependant and features such as speed, resolution and cost are important considerations.

4.2 The ADSP21020

In our case, we have used the ADSP21020 chip, which was available with a minimum of memory usage namely not more than 4 arrays of 128x128 bytes could be used. This processor can perform multifunction computations such as floating point dual add/subtract in the ALU or parallel operation of the multiplier and the ALU. It also provides hardware to implement circular buffers in memory. Each data address generator (DAG) keeps track of up to eight address pointers, eight modifiers, eight buffer length values and eight base values. A pointer used for indirect addressing can be modified by a value in a specified register, either before (pre-modify) or after (post-modify) the access.

To implement automatic modulo addressing for circular buffers, the ADSP-21020 provides buffer length registers that can be associated with each pointer. Base values for pointers allow circular buffers to be placed at arbitrary locations. The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions.

To execute looped code with zero overhead, this chip maintains an internal loop counter and loop stack. No explicit jump or decrement instructions are required to maintain the loop. It derives its high clock rate from pipelined fetch, decode and execute cycles. Approximately 70% of the machine cycle is available for memory accesses; consequently, ADSP-21020 systems can be built using slower and therefore less expensive memory chips.

These features will be very important for the cost-effective use of the DSP cycles for a fast compression, decompression and post-processing of an image. We will now examine these features in relations to real image processing problems and algorithms.

4.3 Two Dimensional DCT for image Processing

The two-dimensional DCT is the corner stone of many image-processing applications. This section provides a detailed analysis of the implementation of the two dimensional DCT using one-dimensional DCT's.

The 2-D FDCT for an 8x8 block as used by the JPEG standard [1], which is slightly different then the standard definition in the sense of a scaling factor is:

$$F(u,v) = \frac{C(u)C(v)}{4}\sum_{j=0}^{7}\sum_{k=0}^{7} f(j,k)\cos\left[\frac{(2j+1)u\pi}{16}\right]\cos\left[\frac{(2k+1)v\pi}{16}\right] \qquad (4.1)$$

And the IDCT as:

$$f(j,k) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\left[\frac{(2j+1)u\pi}{16}\right]\cos\left[\frac{(2k+1)v\pi}{16}\right] \qquad (4.2)$$

Where

$C(0) = \frac{1}{\sqrt{2}}$, $C(n) = 1$, for the other values of n.

The DCT transformation decomposes each input block into a series of waveforms, each with a particular spatial frequency. The 64 waveforms composing the DCT basis functions are depicted in the Fig 4.1. The coefficients, obtained after the FDCT, correspond to the different weights of these waveforms enabling a reconstruction of the original 8x8 matrix through the IDCT

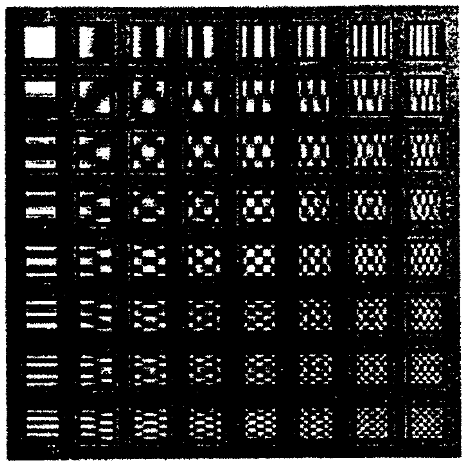

Figure 4.1:The 64 8x8 basis functions

An important property of the 2-D DCT and IDCT is the separability. This property implies that the 2-D DCT can be computed by first performing 1-D DCT's of the rows of the 8x8 matrix, followed by a 1-D DCT of the columns of the obtained matrix. From a computational point of view, each DCT coefficient would require 64 multiplications or a total of 4096 multiply accumulate operations for an 8x8 block. In the next section we will describe a fast 8x8 DCT and IDCT algorithm that reduces considerably this number, while staying very attractive for a hardware implementation.

4.4 The AAN Algorithm

Tseng et al [6] have shown that the 1-D DCT can be solely obtained from the real parts of the DFT having double number of points. Basically, given the eight input values x[k], with k=0,...,7 of the 8-point DCT, the y[k] output values can be calculated as follows:

$$y(n) = \frac{2C(n)\operatorname{Re}Y[n]}{\cos\frac{n\pi}{16}} \tag{4.3}$$

Where

$C(0) = \frac{1}{\sqrt{2}}$, $C(n) = 1$, for the other values of n.

And Y[n] are the output values of 16-point DFT of X[k].

$X[k] = x[k]\ \textit{for}\ \ 0 \le k \le 7$ and $X[k] = x[15-k]\ \textit{for}\ \ 8 \le k \le 15$

One fast way to calculate these Y[n] values has been developed by *Winograd* [7]. In his approach the complex and the real parts are calculated separately. Having noticed that, *Arai, Agui, and Nakajima* [8], designed a flow chart shown in Fig 4.2. that gives the real parts of Y[n] as needed in equation 4.3. Once these values are computed, we only need to scale them to get the explicit values of the coefficients. This is interesting in the JPEG case where the forward 2-D DCT is followed by a quantization or scaling any ways. We can then combine the final scaling factors with the quantization step without making any changes to the arithmetic complexity.

As shown in Fig 4.2. this algorithm needs only 5 multiplications and twenty-nine additions for each coefficient or an equivalent of 80 multiplications and 464 additions for an 8 by 8 pixel block. Another property that makes this algorithm attractive is the symmetry. Indeed this allows a better hardware implementation, as we will see in the next chapter

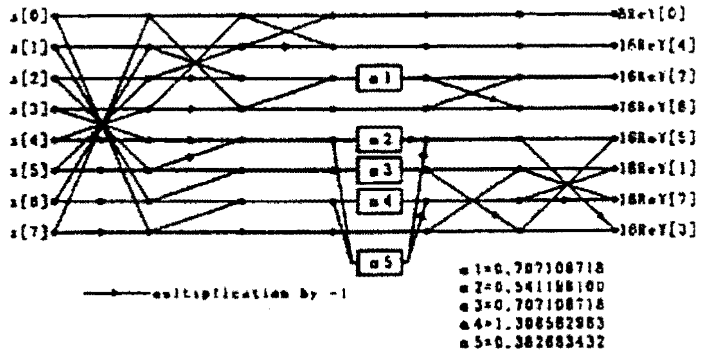

Figure 4.2:Flow chart to find the real parts of 16-point DFT

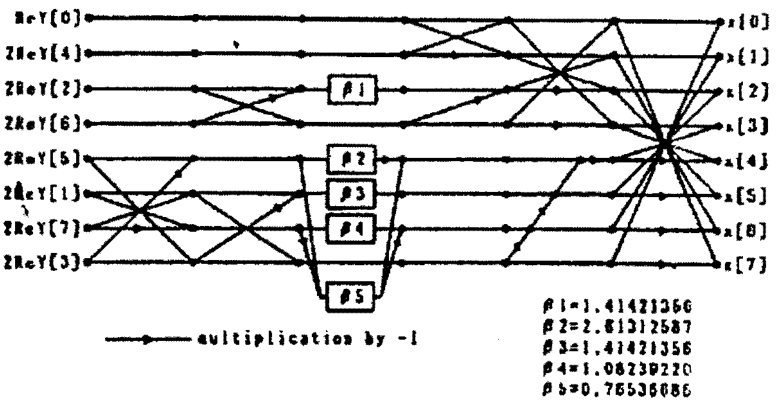

Figure 4.3:Flow chart to restore the original values from the real parts of 16-point DFT

4.5 Hardware Implementation

Using the Ez-lab kit it is possible to simulate the ADSP21020 chip. This chip is located on an evaluation board connected to a PC via a cable. The simulator consists of a C debugger and an interface showing the values of the registers, cycle count, program memory as well as data memory.

We will now show how an efficient assembly redesign can accomplish the task of calculating the forward DCT faster. Since we are dealing with a row/column multiplication of an 8 by 8 matrix, we will separate the code into two parts one for the row transformation and one for the columns. Now that we need to perform the same transformation on each row of the matrix we will use a circular buffer of length 8. The next step is to create a loop with no overhead cycles. This is done by using the assembly instruction "do loop until". We then use a pointer to the first element in the array that will be our base to do the rest of the calculations.

Another feature given by the assembly language is the possibility to "roll" a loop. This means that the operations are pipelined to minimize instructions within a loop. By pipelining we mean exploiting the ADSP-21020's parallel architecture to maximize concurrent operations. Indeed it is possible, with restrictions to specific registers, to use the dual functions in the ALU and a separate data memory access all in one cycle. An example of dual functions is given below:

$$Fa = Fx + Fy, \qquad Fs = Fx - Fy \tag{4.4}$$

Dual Add/subtract

Where Fa, Fs, Fx, Fy are floating point register file locations.

If we go back to the flow graph of the AAN algorithm Fig 4.3 we can see how well it is suited for such dual functions. This is the main reason why we have chosen to implement this algorithm.

All these properties of the ADSP21020 chip made the use of assembly language for all the functions in the code a better solution then having the functions written in C, as the results in the next chapter will prove.

Chapter 5

Tests and Performance Results

5.1 Introduction

In this chapter, we will compare the execution speed of the same algorithm running on a Sparc 5 Sun workstation and on the ADSP21020 chip from Analog Devices. The Sparc 5 was running the code on a 70 Mhz processor of the MicroSparc family. All the hardware and software tests have been done using a 128 by 128 pixels image size because of memory constraints on the ADSP21020. The image is stored in a packed pixel format where four 8-bit pixels reside in a single word. Hence, pixels must be unpacked and converted to floating point numbers before further processing.

5.2 Simulation Algorithm

The algorithm used for comparison purposes is as follows:

1. Read in the original uncompressed image.(128 x 128 pixels)
2. Read in the edge image obtained using the Sobel edge detector.

3. Use JPEG baseline algorithm to compress and decompress the original image at 0.53 bits/pixel
4. Calculate the root mean squared error
5. Start the clock.
6. Perform post-processing
7. End clock
8 Calculate the root mean squared error

5.3 Simulation results

We have used the same C driver function for all the tests to be able to compare results. First the code is compiled on the UNIX machines and the time is taken using the C function clock(void), which returns the processor time used by the program since the beginning of its execution. To measure the time spent during post-processing this function is called before and after it and the last value is subtracted from the first To convert the time in seconds clock( ) / CLOCKS_PER_SEC is used, where CLOCKS_PER_SEC is a constant given by the system.

In the case of the ADSP21020 chip the driver code is ran in both cases with all the functions written in C and with all the functions written in Assembler. The cycle count is taken before and after post-processing. To convert the result to seconds we use the fact that one cycle on this chip is executed in 40ns on the 25 Mhz chip.

Finally we normalize all the timings to 100 Mhz to be able to compare accurately the results. All those tests are done using the AAN algorithm. For comparison purposes, the same code using a straight-forward algorithm executed in 27.91 seconds on a Sparc 5 for a 128x128 image, which is a very long time if we want to implement close to real time post-processing.

	25Mhz	70 Mhz	100Mhz
C code on Sparc 5		4800000/10000000=0.48s	0.33s
C code on DSP chip	27422943 cycles=1.09s		0.27s
Assembly code on DSP	21540386 cycles=0.86s		0.21s

Figure5.1: Execution speed results obtained in seconds

The speed increases we obtain are as follows:

Gain of 21.6% compared to C code on ADSP21020

Gain of 36.3% compared to C code on Sparc 5 workstation.

Chapter 6

Conclusion

In this work we have looked at the implementation aspect of a post-processing algorithm. The main parameter we have considered is the speed. For that purpose, we have implemented this algorithm on different platforms and compared the gains we could obtain by using some optimization.

It is important to note that nowadays JPEG is used for the intra-frame in the video MPEG format. We can then think of this implementation as one fraction of the post-processing that could be used if it is fast enough. The faster we can execute this algorithm, the better it will be for the final video sequence.

Bibliography

[1] William B. Penneaker and Joan L. Mitchell, *Still Image Data Compression Standard*, New York:Van Nostrand Rheinhold (1993)

[2] H.Reeve and J. Lim, "Reduction of blocking effects in image coding," *Optical Engineering*, vol. 23(1), pp 34-37, January/February 1984

[3] W.E. Lynch, A. R. Reihman, and B. Liu, "Edge compensated transform coding," in *Proc. ICIP-94,*pp 105-109, November 1994

[4] J. Luo, C. W. Chen, K. J. Parker, and T.S Huang, "A new method for block effect removal in low bit-rate image compression," in *Proc. ICASSP-94*, vol. V, pp. 341-344, April 1994

[5] W.E. Lynch, A. R. Reihman, and B. Liu, "Post processing transform coded images using edges," in *Proc. ICASSP-95*, pp 2323-2326, May 1995

[6] C.J. Kuo and R. J. Hsieh, "Adaptive postprocessor for block encoded images," IEEE Transactions on Circuits and Systems for Video Technology, vol. 5, pp 298-304, August 1995.

[7] B.D Tseng and W.C.Miller, "On Computing the Discrete Cosine Transform", IEEE Trans. Comput.,vol.10, pp 966-968,October 1978.

[8] S.Winograd, "On Computing the Discrete Fourier Transform", *Mathematics of Computaion.* vol 23,pp 99-175, January 1978.

[9] Y. Arai, T. Agui, and M. Nakajima. A fast DCT-SQ Scheme for images. Trans. of IEICE. vol. E71, pp 1095-1097, November 1988.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define ROWMAX 128
#define COLMAX 128
#define BLOCK_SIZE 8
#define True 1
#define False 0
#define DCTSIZE 8
float scale;
 int Qtable[8][8] = { 16, 11, 10, 16, 24, 40, 51, 61,
                            12, 12, 14, 19, 26, 58, 60, 55,
                            14, 13, 16, 24, 40, 57, 69, 56,
                            14, 17, 22, 29, 51, 87, 80, 62,
                            18, 22, 37, 56, 68,109,103, 77,
                            24, 35, 55, 64, 81,104,113, 92,
                            49, 64, 78, 87,103,121,120,101,
                            72, 92, 95, 98,112,100,103, 99};

float test_matrix[8][8]={0,0,0,1,1,1,1,1,
                       0,0,0,1,1,1,1,1,
                       0,0,0,1,1,1,1,1,
                       1,1,1,1,1,1,1,1,
                       1,1,1,1,1,1,1,1,
                       1,1,1,1,1,1,1,1,
                       1,1,1,1,1,1,1,1,
                       1,1,1,1,1,1,1,1};
static const float aan[BLOCK_SIZE] = {
        1.0, 1.387039, 1.306562, 1.175875,
        1.0, 0.785694, 0.541196, 0.275899
      };


void get_data(int llist[ROWMAX][COLMAX]);
void read_quant_table(int quant[BLOCK_SIZE][BLOCK_SIZE]);
void fdct(int pxs[BLOCK_SIZE][BLOCK_SIZE],float
dctcoef[BLOCK_SIZE][BLOCK_SIZE]);
void quantize(float dctcoef[BLOCK_SIZE][BLOCK_SIZE], int
quant_table[BLOCK_SIZE][BLOCK_SIZE]);
void dequantize(float dctcoef[BLOCK_SIZE][BLOCK_SIZE], int
quant_table[BLOCK_SIZE][BLOCK_SIZE]);
void idct(int pxs[BLOCK_SIZE][BLOCK_SIZE], float
dctcoef[BLOCK_SIZE][BLOCK_SIZE]);
void put_data(int llist_res[ROWMAX][COLMAX]);
float C(int n);
float round(float x);
float error_cal(int pls_in[ROWMAX][COLMAX], int
pls_out[ROWMAX][COLMAX]);
void process(float dct_value[BLOCK_SIZE][BLOCK_SIZE], int Up, int
Down, int Left, int Right, int Up_row[BLOCK_SIZE], int
Down_row[BLOCK_SIZE], int Left_col[BLOCK_SIZE], int
Right_col[BLOCK_SIZE], int nw, int ne, int sw, int se);
```

```
void print(float coef[BLOCK_SIZE][BLOCK_SIZE]);
void printp(int pls[BLOCK_SIZE][BLOCK_SIZE]);
void get_edge_data(int llist[ROWMAX][COLMAX]);
void post_processing(int edge[8][8], int pixels[8][8]);
void set_non_edge_pixels(int u, int v, int segnumber, int
ttemp[10][10]);
float block_rmse(int ori_pixls[8][8], int post_pixls[8][8]);
void square(int res[ROWMAX][COLMAX], int m, int n, int size,
            int msk[ROWMAX][COLMAX]);
void diamond(int res[ROWMAX][COLMAX], int m, int n, int size,
             int msk[ROWMAX][COLMAX]);
void dctprocess(float after[BLOCK_SIZE][BLOCK_SIZE], float
before[BLOCK_SIZE][
BLOCK_SIZE], int quant[BLOCK_SIZE][BLOCK_SIZE]);
void jpeg_fdct_float (float * data);
  void jpeg_idct_float ( float * coef_block)  ;


main()
{
  int noproc[BLOCK_SIZE][BLOCK_SIZE];
  int diff[BLOCK_SIZE][BLOCK_SIZE];
  int mask[ROWMAX][COLMAX];
  int Change;
  int ori[ROWMAX][COLMAX];
  char filename[30];
  int list[ROWMAX][COLMAX];
  int list_res[ROWMAX][COLMAX];
   int listi[ROWMAX][COLMAX];
  float dct1[BLOCK_SIZE][BLOCK_SIZE];
  float pict[BLOCK_SIZE*BLOCK_SIZE];
  int pixels[BLOCK_SIZE][BLOCK_SIZE];
  float dct[BLOCK_SIZE][BLOCK_SIZE];
  int quant[BLOCK_SIZE][BLOCK_SIZE];
  int i, j, k, l, x, y;
  int block_row_max;
  int block_col_max;
  float rmse;
  int m;
  int Up, Down, Left, Right;
  int Up_row[BLOCK_SIZE], Down_row[BLOCK_SIZE];
  int Left_col[BLOCK_SIZE], Right_col[BLOCK_SIZE];
  float detect;
  int mm, nn;
  int edge_list[ROWMAX][COLMAX];
  int edge[BLOCK_SIZE][BLOCK_SIZE];
  int block_num=0;
  int ring_num=0;
  clock_t clo,start;
  time_t tim;
  float wholedct[ROWMAX][COLMAX];
  float olddct[BLOCK_SIZE][BLOCK_SIZE];
  int mark[ROWMAX/BLOCK_SIZE][COLMAX/BLOCK_SIZE];
  int u, v;
```

```
    float dctori[BLOCK_SIZE][BLOCK_SIZE];
    int NW, NE, SW, SE;
  char cmd_string[80];

    printf("Please enter the scale value:\n");
    scanf("%f", &scale);
    printf("Input original data:\n");
    get_data(ori);
   /* printf("Input received data:\n");
    get_data(list);*/

    read_quant_table(quant);

    get_edge_data(edge_list);

    block_row_max = ROWMAX/BLOCK_SIZE;
    block_col_max = COLMAX/BLOCK_SIZE;

   start = clock();


    for (i = 0; i < block_row_max; i++)
    {
       Up = True;
       Down = True;
       if ( i == 0 ) Up = False;
       if ( i == block_row_max - 1 ) Down = False;

       for (j = 0; j < block_col_max; j++)
       {
         Left = True;
         Right = True;
         if ( j == 0 ) Left = False;
         if ( j == block_col_max - 1 ) Right = False;

         if ( Up == True )
           for (m = 0; m < BLOCK_SIZE; m++)
              Up_row[m] =.25 *  list[i*BLOCK_SIZE - 1][j*BLOCK_SIZE
+ m] +
                         .75 *  list[i*BLOCK_SIZE][j*BLOCK_SIZE +
m];

         if ( Down == True )
           for (m = 0; m < BLOCK_SIZE; m++)
              Down_row[m] =.25 *  list[(i +
1)*BLOCK_SIZE][j*BLOCK_SIZE + m] +
                           .75 *  list[(i + 1)*BLOCK_SIZE -
1][j*BLOCK_SIZE + m];

         if ( Left == True )
           for (m = 0; m < BLOCK_SIZE; m++)
              Left_col[m] =.25 * list[i*BLOCK_SIZE + m][j*BLOCK_SIZE
- 1] +
                         .75 * list[i*BLOCK_SIZE +
m][j*BLOCK_SIZE];

       if ( Right == True )
         for (m = 0; m < BLOCK_SIZE; m++)
            Right_col[m] =.25 * list[i*BLOCK_SIZE + m][(j +
1)*BLOCK_SIZE] +
                          .75 * list[i*BLOCK_SIZE + m][(j +
1)*BLOCK_SIZE -1];

       if (Up == True && Left == True)
          NW = list[i*BLOCK_SIZE-1][j*BLOCK_SIZE-1]/16. +
               list[i*BLOCK_SIZE-1][j*BLOCK_SIZE]*3/16. +
               list[i*BLOCK_SIZE][j*BLOCK_SIZE-1]*3/16. +
               list[i*BLOCK_SIZE][j*BLOCK_SIZE]*9/16.;

       if (Up == True && Right == True)
          NE = list[i*BLOCK_SIZE-1][j*BLOCK_SIZE+BLOCK_SIZE]/16. +
               list[i*BLOCK_SIZE-1][j*BLOCK_SIZE+BLOCK_SIZE-
1]*3/16. +
               list[i*BLOCK_SIZE][j*BLOCK_SIZE+BLOCK_SIZE]*3/16. +
               list[i*BLOCK_SIZE][j*BLOCK_SIZE+BLOCK_SIZE-
1]*9/16.;

       if (Down == True && Left == True)
          SW = list[i*BLOCK_SIZE+BLOCK_SIZE][j*BLOCK_SIZE-1]/16. +
               list[i*BLOCK_SIZE+BLOCK_SIZE-1][j*BLOCK_SIZE-
1]*3/16. +
               list[i*BLOCK_SIZE+BLOCK_SIZE][j*BLOCK_SIZE]*3/16. +
               list[i*BLOCK_SIZE+BLOCK_SIZE-
1][j*BLOCK_SIZE]*9/16.;

       if (Down == True && Right == True)
          SE =
list[i*BLOCK_SIZE+BLOCK_SIZE][j*BLOCK_SIZE+BLOCK_SIZE]/16. +
             list[i*BLOCK_SIZE+BLOCK_SIZE-
1][j*BLOCK_SIZE+BLOCK_SIZE]*3/16. +

list[i*BLOCK_SIZE+BLOCK_SIZE][j*BLOCK_SIZE+BLOCK_SIZE-1]*3/16. +
             list[i*BLOCK_SIZE+BLOCK_SIZE-
1][j*BLOCK_SIZE+BLOCK_SIZE-1]*9/16.;

       x = 0;
       for (k = i * BLOCK_SIZE; k < (i + 1) * BLOCK_SIZE; k++)
       {
          y = 0;
          for (l = j * BLOCK_SIZE; l < (j + 1) * BLOCK_SIZE; l++)
          {
             pixels[x][y] = ori[k][l];
                pict[x*BLOCK_SIZE+y] = (float)(ori[k][l]-128);
             y++;
          }
          x++;
       }
```

```
jpeg_fdct_float (pict);

              for (u = 0; u < BLOCK_SIZE; u++)
          for (v = 0; v < BLOCK_SIZE; v++)
               {
                       dct[u][v] = pict[u*
BLOCK_SIZE+v]/(aan[u]*aan[v]*8);

            }

/* Quantize DCT Coefficient */
       quantize(dct, quant);

/* Dequantize the data */
       dequantize(dct, quant);

for (u = 0; u < BLOCK_SIZE; u++)
          for (v = 0; v < BLOCK_SIZE; v++)
               {
                        pict[u*
BLOCK_SIZE+v]=dct[u][v]*(aan[u]*aan[v]*8);

            }

for (u = 0; u < BLOCK_SIZE; u++)
          for (v = 0; v < BLOCK_SIZE; v++)
              dctori[u][v] = dct[u][v];



       mark[i][j] = 1;
       for (x = 0; x < BLOCK_SIZE; x++)
       {
         if (mark[i][j] == 0) break;
         for (y = 0; y < BLOCK_SIZE; y++)
           if ( ((x+y)>1) && (dct[x][y] != 0.) )
           {
              mark[i][j] = 0;
              break;
           }
       }









       Change = True;
       for (x = 0; x < BLOCK_SIZE; x++)
         for (y = 0; y < BLOCK_SIZE; y++)
```

```
            if ( ((x+y)>1) && (dct[x][y] != 0.) )
            {
               Change = False;
               break;
            }


jpeg_idct_float (pict);

        for (x = 0; x < BLOCK_SIZE; x++)
        {
           for (y = 0; y < BLOCK_SIZE; y++)
           {
              if (pixels[x][y] > 255) pixels[x][y] = 255;
              if (pixels[x][y] < 0) pixels[x][y] = 0;
           if ((pict[x* BLOCK_SIZE+y]+128.5) < 0){
   pict[x* BLOCK_SIZE+y] = -128;
   }
     list_res[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] =
pict[x*BLOCK_SIZE+y]+128.5;
wholedct[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] = dctori[x][y];
              if (mark[i][j] == 1)
                  mask[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] = 0;
              else
                  mask[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] = 1;
              if (Change == True)
                  list[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] = 0;
           }
        }
      }
   }
clo =clock();
printf("the elapsed time is %ld",(clo-start));
printf("Input the rec file name:\n");
put_data(list_res);

printf("First Step:\n");
rmse = error_cal(ori, list_res);
printf("The root mean-squared error is %f\n", rmse);

start =clock();
  for (i = 0; i < block_row_max; i++)
  {
     for (j = 0; j < block_col_max; j++)
     {
       x = 0;
       for (k = i * BLOCK_SIZE; k < (i + 1) * BLOCK_SIZE; k++)
       {
          y = 0;
          for (l = j * BLOCK_SIZE; l < (j + 1) * BLOCK_SIZE; l++)
          {
             pixels[x][y] = list_res[k][l];
             edge[x][y] = edge_list[k][l];
             dct[x][y] = wholedct[k][l];
```

```
              y++;
           }
           x++;
         }


         detect = 0.;
         for (mm = 0; mm < BLOCK_SIZE; mm++)
            for (nn = 0; nn < BLOCK_SIZE; nn++)
              detect = detect + abs(dct[mm][nn]) *
test_matrix[mm][nn];


         if (detect > 0.)
         {
           ring_num++;
           post_processing(edge, pixels);
         }

         for (x = 0; x < BLOCK_SIZE; x++)
         {
            for (y = 0; y < BLOCK_SIZE; y++)
            {
               if (pixels[x][y] > 255) pixels[x][y] = 255;
               if (pixels[x][y] < 0) pixels[x][y] = 0;
               list_res[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] =
pixels[x][y];
               if (detect > 0.) list[i * BLOCK_SIZE + x][j *
BLOCK_SIZE + y] = 0;
            }
         }
      }
   }

square(list_res, 4, 4, 8, mask);


   for (i = 0; i < block_row_max; i++)
   {
      for (j = 0; j < block_col_max; j++)
      {
        x = 0;
        for (k = i * BLOCK_SIZE; k < (i + 1) * BLOCK_SIZE; k++)
        {
           y = 0;
           for (l = j * BLOCK_SIZE; l < (j + 1) * BLOCK_SIZE; l++)
           {
              pixels[x][y] = list_res[k][l];
              olddct[x][y] = wholedct[k][l];
              pict[x*BLOCK_SIZE+y] = (float)(list_res[k][l]-128);
              y++;
           }
           x++;
        }
```

```
              jpeg_fdct_float (pict);
 for (u = 0; u < BLOCK_SIZE; u++)
           for (v = 0; v < BLOCK_SIZE; v++)
                                dct[u][v] =
pict[u*BLOCK_SIZE+v]/(aan[u]*aan[v]*8);
        dctprocess(dct, olddct, quant);
 for (u = 0; u < BLOCK_SIZE; u++)
           for (v = 0; v < BLOCK_SIZE; v++)
               pict[u* BLOCK_SIZE+v]=dct[u][v]*(aan[u]*aan[v]*8);
       jpeg_idct_float (pict);


        for (x = 0; x < BLOCK_SIZE; x++)
        {
           for (y = 0; y < BLOCK_SIZE; y++)
           {
              if (pixels[x][y] > 255) pixels[x][y] = 255;
              if (pixels[x][y] < 0) pixels[x][y] = 0;
 if ((pict[x* BLOCK_SIZE+y]+128.5) < 0){
   pict[x* BLOCK_SIZE+y] = -128;

  }
     list_res[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] =
pict[x*BLOCK_SIZE+y]+128.5;
           /*  list_res[i * BLOCK_SIZE + x][j * BLOCK_SIZE + y] =
pixels[x][y];*/
           }
        }
     }
  }
clo =clock();
printf("the elapsed time in %ld is %ld,%ld",CLOCKS_PER_SEC,(clo-
start),(clo-start));
  printf("Input the data file name:\n");
  put_data(list_res);

/* Calculate the root mean-squared error */
  printf("Fourth Step:\n");
  rmse = error_cal(ori, list_res);
  printf("The root mean-squared error is %f\n", rmse);

  printf("# of blocking block is %d\n", block_num);
  printf("# of ringing block is %d\n", ring_num);
}

void get_data(int llist[ROWMAX][COLMAX])
{
  int data, j, k;
  FILE *fd;
  char filename[30];

  printf("Please enter the file name to be read:\n");
  scanf("%s", filename);
  fd = fopen(filename, "r");
  if (fd != NULL)
  {
    for (k = 0; k < ROWMAX; k++)
    {
       for (j = 0; j < COLMAX; j++)
        {
           if (fscanf(fd, "%d", &data) != EOF)
           {
             llist[k][j] = data;
           }
        }
    }
  }
  else
    printf("File not found or is empty.");
  fclose(fd);
}

void read_quant_table(int qtable[BLOCK_SIZE][BLOCK_SIZE])
{
  int data, j, k;
  FILE *fd;
  char filename[30];

  printf("Please enter the quantization table file name to be
read:\n");
  scanf("%s", filename);
  fd = fopen(filename, "r");
  if (fd != NULL)
  {
    for (k = 0; k < BLOCK_SIZE; k++)
    {
       for (j = 0; j < BLOCK_SIZE; j++)
       {
          if (fscanf(fd, "%d", &data) != EOF)
          {
             qtable[k][j] = data;
          }
       }
    }
  }
  else
    printf("File not found or is empty.");
  fclose(fd);
}

void fdct(int pxs[BLOCK_SIZE][BLOCK_SIZE], float
dctcoef[BLOCK_SIZE][BLOCK_SIZE])
{
  short x, y, u, v;
  float sum;

  for (u =0; u < 8; u++)
```

```
      for (v = 0; v < 8; v++)
      {
      sum = 0.;

      for (x = 0; x < 8; x++)
         for (y = 0; y < 8; y++)
            sum += (pxs[x][y]-128) * cos((2*x+1)*u*(3.1415926/16.))
* cos((2*y+                1)*v*(3.1415926/16.));
      dctcoef[u][v] = 0.25 * C(u)*C(v) * sum;
     }
}

void idct(int pxs[BLOCK_SIZE][BLOCK_SIZE], float
dctcoef[BLOCK_SIZE][BLOCK_SIZE])
{
  short x, y, u, v;
  float sum;

  for (x = 0; x < 8; x++)
      for (y = 0; y < 8; y++)
      {
      sum = 0.;

      for (u =0; u < 8; u++)
         for (v = 0; v < 8; v++)
           sum += C(u) * C(v) * dctcoef[u][v] *
cos((2*x+1)*u*(3.1415926/16.)) *
cos((2*y+1)*v*(3.1415926/16.));
      pxs[x][y] =  128.5+0.25 * sum;
      }
}

float C(int n)
{
   if ( n == 0) return (1./sqrt(2.));
   else return(1.);
}

void quantize(float dctcoef[BLOCK_SIZE][BLOCK_SIZE], int
quant_table[BLOCK_SIZE][BLOCK_SIZE])
{
   short i, j;
   for (i = 0; i < 8; i++)
      for(j = 0; j < 8; j++)

dctcoef[i][j]=round(dctcoef[i][j]/quant_table[i][j]/scale);
}

void dequantize(float dctcoef[BLOCK_SIZE][BLOCK_SIZE], int
quant_table[BLOCK_SIZE][BLOCK_SIZE])
{
   short i, j;
   for (i = 0; i < 8; i++)
      for(j = 0; j < 8; j++)
```

```
          dctcoef[i][j]=dctcoef[i][j]*quant_table[i][j]*scale;
}

float round(float x)
{
   if (x > 0.) return ((int) (x+0.5));
   else return ((int) (x-0.5));
}

void put_data(int llist_res[ROWMAX][COLMAX])
{
   int j,k;
   char filename[30];
   FILE *fd;

   printf( "Please enter the file name to be written to:\n");
   scanf("%s", filename);
   fd= fopen( filename, "w" );
   fprintf(fd, "P2\n%d %d\n255\n", ROWMAX, COLMAX);
   for (j=0; j< ROWMAX; j++)
   {
      for (k=0; k< COLMAX; k++)
         fprintf(fd, "%3d ", llist_res[j][k]);
      fprintf(fd, "\n");
   }
   fclose( fd );
}

float error_cal(int pls_in[ROWMAX][COLMAX], int
pls_out[ROWMAX][COLMAX])
{
  int i, j;
  float sum;
  float error;

  sum = 0.;
  for (i = 0; i < ROWMAX; i++)
     for (j = 0; j< COLMAX; j++)
        sum += (pls_in[i][j] - pls_out[i][j]) * (pls_in[i][j] -
pls_out[i][j]);
  error = sqrt(sum/ROWMAX/COLMAX);

  return (error);
}

void process(float dct_value[BLOCK_SIZE][BLOCK_SIZE], int Up, int
Down, int Left, int Right, int Up_row[BLOCK_SIZE], int
Down_row[BLOCK_SIZE], int Left_col[BLOCK_SIZE], int
Right_col[BLOCK_SIZE], int nw, int ne, int sw, int se)
{
  float temp[BLOCK_SIZE][BLOCK_SIZE];
  int pel[BLOCK_SIZE][BLOCK_SIZE];
  int i,j,iterate;
```

```
for (i = 0; i < BLOCK_SIZE; i++)
   for (j = 0; j < BLOCK_SIZE; j++)
   {
      temp[i][j] = dct_value[i][j];
   }

for (iterate = 0; iterate < 1; iterate++)
{
   idct(pel, dct_value);

   for (j = 1; j < BLOCK_SIZE-1; j++)
   {
     if ( Up == True )
        pel[0][j] = Up_row[j];
     if ( Down == True )
        pel[7][j] = Down_row[j];
   }
   for (i = 1; i < BLOCK_SIZE-1; i++)
   {
     if (Left == True )
        pel[i][0] = Left_col[i];
     if ( Right == True )
        pel[i][7] = Right_col[i];
   }

   if ( Up == True && Left == True )
      pel[0][0] = nw;
   else if (Up == True)
      pel[0][0] = Up_row[0];
   else if (Left == True)
      pel[0][0] = Left_col[0];

   if ( Up == True && Right == True )
      pel[0][7] = ne;
   else if (Up == True)
      pel[0][7] = Up_row[7];
   else if (Right == True)
      pel[0][7] = Right_col[0];

   if ( Down == True && Left == True )
      pel[7][0] = sw;
   else if (Down == True)
      pel[7][0] = Down_row[0];
   else if (Left == True)
      pel[7][0] = Left_col[7];

   if ( Down == True && Right == True)
      pel[7][7] = se;
   else if (Down == True)
      pel[7][7] = Down_row[7];
   else if (Right == True)
      pel[7][7] = Right_col[7];

   fdct(pel, dct_value);
/*      for (i = 0; i < BLOCK_SIZE; i++)
         for (j = 0; j < BLOCK_SIZE; j++)
         {
           if (abs(temp[i][j]) != 0.) dct_value[i][j] = temp[i][j];
         }        */
  }
}

void print(float coef[BLOCK_SIZE][BLOCK_SIZE])
{
  short i;
  putchar('\n');
  for (i=0; i<8; i++)
    printf(
      "Row %d: %5.1lf %5.1lf %5.1lf %5.1lf"
      " %5.1lf %5.1lf %5.1lf %5.1lf\n",
      i, coef[i][0], coef[i][1], coef[i][2], coef[i][3],
      coef[i][4], coef[i][5], coef[i][6], coef[i][7]);
}

void printp(int pls[BLOCK_SIZE][BLOCK_SIZE])
{
  short i;
  putchar('\n');
  for (i=0; i<8; i++)
    printf(
     "Row %d: %3d %3d %3d %3d %3d %3d %3d %3d\n",
     i, pls[i][0], pls[i][1], pls[i][2], pls[i][3],
     pls[i][4], pls[i][5], pls[i][6], pls[i][7]);
}

void get_edge_data(int llist[ROWMAX][COLMAX])
{
  int data, j, k;
  FILE *fd;
  char filename[30];

  printf("Please enter the edge file name to be read:\n");
  scanf("%s", filename);
  fd = fopen(filename, "r");
  if (fd != NULL)
  {
    for (k = 0; k < ROWMAX; k++)
    {
       for (j = 0; j < COLMAX; j++)
        {
           if (fscanf(fd, "%d", &data) != EOF)
           {
             llist[k][j] = data;
           }
        }
    }
  }
  else
```

```
    printf("File not found or is empty.");
  fclose(fd);
}

void post_processing(int edge[8][8], int pixels[8][8])
{
  int temp[10][10];
  int seg_num = 1;
  int x, y, i;
  long int sum = 0;
  int count = 0;
  int averg;

  x = 0;
  for (y = 0; y < 10; y++)  temp[x][y] = 255;
  x = 9;
  for (y = 0; y < 10; y++)  temp[x][y] = 255;
  y = 0;
  for (x = 0; x < 10; x++)  temp[x][y] = 255;
  y = 9;
  for (x = 0; x < 10; x++)  temp[x][y] = 255;

  for (x = 1; x < 9; x++)
    for (y = 1; y < 9; y++)
       temp[x][y] = edge[x-1][y-1];

  for (x = 1; x < 9; x++)
    for (y = 1; y < 9; y++)
       if (temp[x][y] == 0)
       {
         temp[x][y] = seg_num;
         set_non_edge_pixels(x, y, seg_num, temp);
         seg_num++;
        }

  for (i = 1; i < seg_num; i++)
  {
    for (x = 1; x < 9; x++)
       for (y = 1; y < 9; y++)
          if (temp[x][y] == i)
          {
            sum = sum + pixels[x-1][y-1];
            count++;
          }
    averg = sum/count;
    for (x = 1; x < 9; x++)
       for (y = 1; y < 9; y++)
          if (temp[x][y] == i)
             pixels[x-1][y-1] = averg;
    sum = 0;
    count = 0;
  }
}
```

```
void set_non_edge_pixels(int u, int v, int segnumber, int
ttemp[10][10])
{
  if (ttemp[u-1][v] == 0)
    {
      ttemp[u-1][v] = segnumber;
      set_non_edge_pixels(u-1, v, segnumber, ttemp);
    }
  if (ttemp[u+1][v] == 0)
    {
      ttemp[u+1][v] = segnumber;
      set_non_edge_pixels(u+1, v, segnumber, ttemp);
    }
  if (ttemp[u][v-1] == 0)
    {
      ttemp[u][v-1] = segnumber;
      set_non_edge_pixels(u, v-1, segnumber, ttemp);
    }
  if (ttemp[u][v+1] == 0)
    {
      ttemp[u][v+1] = segnumber;
      set_non_edge_pixels(u, v+1, segnumber, ttemp);
    }
}

void square(int res[ROWMAX][COLMAX], int m, int n, int size,
            int msk[ROWMAX][COLMAX])
{
  int a, b, c, d;
  int i,j;

  for(i = m; i < ROWMAX - size; i = i + size)
   for(j = n; j < COLMAX - size; j = j + size)
   {
    a = res[i][j];
    b = res[i][j+size];
    c = res[i+size][j+size];
    d = res[i+size][j];

    if (msk[i][j] == 0 && msk[i][j+size] == 0 &&
msk[i+size][j+size] == 0
        && msk[i+size][j] == 0)
       res[i+size/2][j+size/2] = (a+b+c+d)/4;
   }
}

void diamond(int res[ROWMAX][COLMAX], int m, int n, int size,
            int msk[ROWMAX][COLMAX])
{
  int a, b, c, d;
  int i, j;
  for(i = m; i < ROWMAX - size/2; i = i + size)
   for(j = n; j < COLMAX - size; j = j + size)
   {
     a = res[i][j];
     b = res[i-size/2][j+size/2];
     c = res[i][j+size];
     d = res[i+size/2][j+size/2];

     if (msk[i][j] == 0 && msk[i-size/2][j+size/2] == 0 &&
msk[i][j+size] == 0
         && msk[i+size/2][j+size/2] == 0)
           res[i][j+size/2] = (a+b+c+d)/4;
   }
}

void dctprocess(float after[BLOCK_SIZE][BLOCK_SIZE],
                float before[BLOCK_SIZE][BLOCK_SIZE],
                int qtable[BLOCK_SIZE][BLOCK_SIZE])
{
   int i,j;

   for (i = 0; i < BLOCK_SIZE; i++)
     for (j = 0; j < BLOCK_SIZE; j++)
     {
       if (abs(after[i][j]) > (abs(before[i][j]) + scale *
qtable[i][j]/2.))
       {
         if (after[i][j] > 0.)
            after[i][j] = abs(before[i][j]) + scale *
qtable[i][j]/2.;
         else
            after[i][j] = -(abs(before[i][j]) + scale
*qtable[i][j]/2.);
       }
       else if (abs(after[i][j]) < (abs(before[i][j]) - scale
*qtable[i][j]/2.))
       {
         if (after[i][j] > 0.)
            after[i][j] = abs(before[i][j]) - scale *
qtable[i][j]/2.;
         else
            after[i][j] = -(abs(before[i][j]) - scale *
qtable[i][j]/2.);
       }
     }
}
void jpeg_fdct_float (float * data)
{
  float tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  float tmp10, tmp11, tmp12, tmp13;
  float z1, z2, z3, z4, z5, z11, z13;
  float *dataptr;
  int ctr;

  /* Pass 1: process rows. */
  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[0] + dataptr[7];
    tmp7 = dataptr[0] - dataptr[7];
    tmp1 = dataptr[1] + dataptr[6];
    tmp6 = dataptr[1] - dataptr[6];
    tmp2 = dataptr[2] + dataptr[5];
    tmp5 = dataptr[2] - dataptr[5];
    tmp3 = dataptr[3] + dataptr[4];
    tmp4 = dataptr[3] - dataptr[4];

    /* Even part */

    tmp10 = tmp0 + tmp3;        /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[0] = tmp10 + tmp11; /* phase 3 */
    dataptr[4] = tmp10 - tmp11;

    z1 = (tmp12 + tmp13) * ((float) 0.707107); /* c4 */
    dataptr[2] = tmp13 + z1;    /* phase 5 */
    dataptr[6] = tmp13 - z1;

    /* Odd part */

    tmp10 = tmp4 + tmp5;        /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

    /* The rotator is modified from fig 4-8 to avoid extra
negations. */
    z5 = (tmp10 - tmp12) * ((float) 0.382683); /* c6 */
    z2 = ((float) 0.541196) * tmp10 + z5; /* c2-c6 */
    z4 = ((float) 1.306562) * tmp12 + z5; /* c2+c6 */
  z3 = tmp11 * ((float) 0.707107); /* c4 */

    z11 = tmp7 + z3;            /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[5] = z13 + z2;      /* phase 6 */
    dataptr[3] = z13 - z2;
    dataptr[1] = z11 + z4;
    dataptr[7] = z11 - z4;
    dataptr += DCTSIZE;         /* advance pointer to next row */
  }

  /* Pass 2: process columns. */

  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
    tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
    tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
    tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
    tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
    tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
    tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
    tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];

    /* Even part */

    tmp10 = tmp0 + tmp3;        /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[DCTSIZE*0] = tmp10 + tmp11; /* phase 3 */
    dataptr[DCTSIZE*4] = tmp10 - tmp11;

    z1 = (tmp12 + tmp13) * ((float) 0.707107); /* c4 */
    dataptr[DCTSIZE*2] = tmp13 + z1; /* phase 5 */
    dataptr[DCTSIZE*6] = tmp13 - z1;

    /* Odd part */

    tmp10 = tmp4 + tmp5;        /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

  /* The rotator is modified from fig 4-8 to avoid extra
negations. */
    z5 = (tmp10 - tmp12) * ((float) 0.382683); /* c6 */
    z2 = ((float) 0.541196) * tmp10 + z5; /* c2-c6 */
    z4 = ((float) 1.306562) * tmp12 + z5; /* c2+c6 */
    z3 = tmp11 * ((float) 0.707107); /* c4 */

    z11 = tmp7 + z3;            /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[DCTSIZE*5] = z13 + z2; /* phase 6 */
    dataptr[DCTSIZE*3] = z13 - z2;
    dataptr[DCTSIZE*1] = z11 + z4;
    dataptr[DCTSIZE*7] = z11 - z4;

    dataptr++;                  /* advance pointer to next column
*/
  }
}

void jpeg_idct_float ( float * coef_block)

{
  float tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  float tmp10, tmp11, tmp12, tmp13;
  float z5, z10, z11, z12, z13;
  float * inptr;
  int ctr;
  float workspace[DCTSIZE*DCTSIZE]; /* buffers data between passes
*/


  /* Pass 1: process columns from input, store into work array. */

  inptr = coef_block;


  for (ctr = DCTSIZE; ctr > 0; ctr--) {


         /* Even part */

    tmp0 = inptr[DCTSIZE*0];
    tmp1 = inptr[DCTSIZE*2];
    tmp2 = inptr[DCTSIZE*4];
    tmp3 = inptr[DCTSIZE*6];

    tmp10 = tmp0 + tmp2;        /* phase 3 */
    tmp11 = tmp0 - tmp2;

    tmp13 = tmp1 + tmp3;        /* phases 5-3 */
    tmp12 = (tmp1 - tmp3) * ((float) 1.414213) - tmp13; /* 2*c4 */

    tmp0 = tmp10 + tmp13;       /* phase 2 */
    tmp3 = tmp10 - tmp13;
 tmp1 = tmp11 + tmp12;
    tmp2 = tmp11 - tmp12;

    /* Odd part */

    tmp4 =inptr[DCTSIZE*1];
    tmp5 =inptr[DCTSIZE*3];
    tmp6 =inptr[DCTSIZE*5];
    tmp7 =inptr[DCTSIZE*7];
 z13 = tmp6 + tmp5;          /* phase 6 */
    z10 = tmp6 - tmp5;
    z11 = tmp4 + tmp7;
    z12 = tmp4 - tmp7;

    tmp7 = z11 + z13;           /* phase 5 */
    tmp11 = (z11 - z13) * ((float) 1.414213); /* 2*c4 */

    z5 = (z10 + z12) * ((float) 1.847759); /* 2*c2 */
    tmp10 = ((float) 1.082392) * z12 - z5; /* 2*(c2-c6) */
    tmp12 = ((float) -2.613125) * z10 + z5; /* -2*(c2+c6) */

    tmp6 = tmp12 - tmp7;        /* phase 2 */
    tmp5 = tmp11 - tmp6;
    tmp4 = tmp10 + tmp5;
    inptr[DCTSIZE*0] = (tmp0 + tmp7)/8;
    inptr[DCTSIZE*7] = (tmp0 - tmp7)/8;
    inptr[DCTSIZE*1] = (tmp1 + tmp6)/8;
    inptr[DCTSIZE*6] = (tmp1 - tmp6)/8;
    inptr[DCTSIZE*2] = (tmp2 + tmp5)/8;
    inptr[DCTSIZE*5] = (tmp2 - tmp5)/8;
    inptr[DCTSIZE*4] = (tmp3 + tmp4)/8;
    inptr[DCTSIZE*3] = (tmp3 - tmp4)/8;

    inptr++;              /* advance pointers to next column */


  }

  /* Pass 2: process rows from work array, store into output
array. */
  /* Note that we must descale the results by a factor of 8 ==
2**3. */

  inptr = coef_block;
  for (ctr = 0; ctr < DCTSIZE; ctr++) {

   /* Even part */

    tmp10 = inptr[0] + inptr[4];
    tmp11 = inptr[0] - inptr[4];

    tmp13 = inptr[2] + inptr[6];
    tmp12 = (inptr[2] - inptr[6]) * ((float) 1.414213) - tmp13;

    tmp0 = tmp10 + tmp13;
    tmp3 = tmp10 - tmp13;
    tmp1 = tmp11 + tmp12;
    tmp2 = tmp11 - tmp12;

    /* Odd part */

    z13 = inptr[5] + inptr[3];
    z10 = inptr[5] - inptr[3];
    z11 = inptr[1] + inptr[7];
    z12 = inptr[1] - inptr[7];
 tmp7 = z11 + z13;
    tmp11 = (z11 - z13) * ((float) 1.414213);

    z5 = (z10 + z12) * ((float) 1.847759); /* 2*c2 */
    tmp10 = ((float) 1.082392) * z12 - z5; /* 2*(c2-c6) */
    tmp12 = ((float) -2.613125) * z10 + z5; /* -2*(c2+c6) */

    tmp6 = tmp12 - tmp7;
    tmp5 = tmp11 - tmp6;
    tmp4 = tmp10 + tmp5;

    inptr[0] = (tmp0 + tmp7)/8;
    inptr[7] = (tmp0 - tmp7)/8;
    inptr[1] = (tmp1 + tmp6)/8;
    inptr[6] = (tmp1 - tmp6)/8;
    inptr[2] = (tmp2 + tmp5)/8;
    inptr[5] = (tmp2 - tmp5)/8;
    inptr[4] = (tmp3 + tmp4)/8;
    inptr[3] = (tmp3 - tmp4)/8;


    inptr += DCTSIZE;           /* advance pointer to next row */
  }

}
```

```
#include "asm_sprt.h"


.segment /pm seg_pmco ;

.global _dct8x8   ;

_dct8x8:
        leaf_entry ;

       !  Save the registers

        pushm(m0) ;
        pushm(m1) ;
        pushm(m2) ;
        pushi(i0) ;
        pushi(i1) ;
        pushi(i2) ;
        pushr(r1) ;
        pushr(r2) ;
        pushr(r3) ;
        pushr(r5) ;
        pushr(r6) ;
        pushr(r7) ;
        pushr(r9) ;
        pushr(r10) ;
        pushr(r11) ;

        i0 = r4  ;
        b2 = const ;
        r12 = 64 ;
        l2  = r12 ;
        r12 = 8 ;
        m0  = r12 ;
        m1=32;
        m2=48;
        i4  = i0 ;
        i1  = i0 ;
        r8 = pass r8 ;

        if eq jump inverse ;
        l1  = r12 ;
        lcntr = r12, do floop1 until lce ;

        b1 = i0 ;
        f0=dm(i1,1);
        f1=dm(i1,1);
        f2=dm(i1,1);
        f3=dm(i1,1);
        f4=dm(i1,1);
        f12=f3+f4, f8=f3-f4, f5=dm(i1,1);
        f9=f2+f5, f5=f2-f5, f6=dm(i1,1);
        f14=f1+f6, f6=f1-f6, f7=dm(i1,1);
        f15=f0+f7, f7=f0-f7              ;
        f10=f15+f12, f13=f15-f12;
        f11=f14+f9 , f12=f14-f9;
        f0=f10+f11, f1=f10-f11,f4 = dm(i2,0);

        f9=f12+f13, dm(i1,1)=f0;
        f9=f9*f4   , dm(3,i1)=f1;
        f2=f13+f9, f3=f13-f9;
        f10=f8+f5, dm(1,i1)=f2;
        f11=f5+f6, dm(5,i1)=f3;

        f12= f6 + f7;
        f5 = f10 - f12,f0=dm(1,i2);
        f5 = f5*f0,f0=dm(2,i2);
        f2 = f10*f0 ;
        f2 = f2 + f5,f0=dm(3,i2);
        f4 = f12*f0;
        f4 = f4 + f5,f0=dm(0,i2);
        f3 = f11*f0;
        f11=f7+f3   ,f13=f7-f3;
        f5 =f13+f2 ,f3 =f13-f2;
        f1 =f11+f4 ,f7 =f11-f4, dm(4,i1) = f5 ;
        dm(2, i1) = f3 ;
        dm(0, i1) = f1 ;
        dm(6, i1) = f7 ;

floop1: modify(i0, m0) ;

       r12=8;
       i0=i4;
       l1= 12;
       lcntr = r12, do floop2 until lce;
       b1 = i0;
```

```
 f0=dm(i1,m0);
 f1=dm(i1,m0);
 f2=dm(i1,m0);
 f3=dm(i1,m0);
 f4=dm(i1,m0);
 f12=f3+f4, f8=f3-f4, f5=dm(i1,m0);
 f9=f2+f5, f5=f2-f5, f6=dm(i1,m0);
 f14=f1+f6, f6=f1-f6, f7=dm(i1,m0);
 f15=f0+f7, f7=f0-f7               ;

 f10 = f15 +f12,f13 = f15-f12;
 f11 = f14+f9 ,f12 = f14 - f9;

 f0=f10+f11,f1=f10-f11;

 dm (i1,m1) = f0;

 dm (i1,m2) = f1;



 f1= f12 + f13,f0=dm(0,i2);
 f1= f1 * f0;
 f2=f13 + f1,f0=f13 - f1;




 f10 = f8 + f5,dm  (i1,m1) = f2;
 f11 = f5 + f6, dm ( i1, m0) = f0;
 f12 = f6 + f7 ;

 f5 = f10 - f12,f0=dm(1,i2) ;

 f5 = f5*f0,f0=dm(2,i2);

 f2 = f10*f0 ;
 f2 = f2 + f5,f0=dm(3,i2);

 f4 = f12*f0;
 f4 = f4 + f5,f0=dm(0,i2);

 f3 = f11*f0;

 f11 = f7 + f3,f13 = f7 -f3;
  f1 = f11 + f4,f7= f11 -f4;
 f5= f13 + f2 ,f3= f13 - f2,  dm(i1,16) = f7 ;



dm(i1,16) = f1 ;
```

```
       dm(i1,16) = f3 ;
       dm(i1,16) = f5 ;

floop2: modify(i0, 1) ;
        jump cas0 ;

inverse:
        r8  = 64 ;
        i4  = i0 ;
        l1  = 12 ;
        m2 =16;
        m3 = 24;

       lcntr = r12, do iloopl until lce ;

       b1 = i0 ;
       f0 = dm (i1,m2);
       f1 = dm (i1,m2);
       f2 = dm (i1,m2) ;
       f10 = f0 + f2, f11 = f0 - f2, f3 = dm ( i1, m3)  ;
       f13 = f1 + f3, f12 = f1 - f3,f15=dm(4,i2);
       f12 = f12* f15;
       f12 = f12 - f13,f4 = dm ( i1,m2);
       f8 = f10 +f13,f3 = f10 - f13,f5= dm (i1,m2);
       f1 = f11 + f12,f2= f11 - f12,f6=dm ( i1,m2) ;
       f13 = f6 + f5, f10 = f6 -f5,f7= dm (i1,m0) ;
       f0=f10;
       f11 = f4 + f7, f12 = f4 -f7;
       f7 = f11 + f13, f11 = f11 - f13;
       f11 = f11 * f15;
       f9 = f10 + f12,f15=dm(5,i2) ;
       f9 = f9 * f15,f15=dm(6,i2);
       f10 = f12*f15 ;
       f10 = f10 - f9,f15=dm(7,i2);
       f12 = f0*f15;
       f12 =f12 + f9;
       f6 = f12  - f7;
       f5 = f11 - f6;
       f4 = f10 + f5;
       f9 = f8;
       f0 = f8 + f7,r15=dm(8,i2);
       f0 = scalb f0 by r15;
       f8 = f1 + f6, dm ( i1,m0 ) =f0;
       f8 = scalb f8 by r15;
       f8 = f2 + f5,dm ( i1, m0 ) = f8;
       f8 = scalb f8 by r15;
       f8 = f3 - f4,dm (i1,m0) =f8;
       f8 = scalb f8 by r15;
       f4 = f3 + f4,dm ( i1 ,m0 ) = f8 ;
       f4 = scalb f4 by r15;
       f5 = f2 - f5, dm (i1,m0) =  f4 ;
       f5 = scalb f5 by r15;
       f6 = f1 - f6, dm (i1,m0) =  f5;
       f6 = scalb f6 by r15;
```

```
        f7 = f9 - f7, dm (i1,m0) = f6;
        f7= scalb f7 by r15;
        dm (i1,m0) =  f7;

iloop1: modify(i0, 1) ;

        i0 = i4 ;
        r12 = 8;
        l1 = r12 ;
        lcntr = r12, do iloop2 until lce ;

        b1 = i0 ;
        f0= dm(i1,1) ;
        f1 = dm (i1,1);
        f2 = dm (i1,1);
        f3 = dm (i1 ,1 );
        f4 = dm ( i1 ,1 ) ;
        f5 = dm (i1 ,1 );
        f6 = dm ( i1,1 ) ;
        f14 = f3;
        f15 = f1;
        f10 = f0 + f4,f11 = f0 - f4,f7 = dm (i1, 1);
        f13 = f2 + f6, f12 = f2 - f6,f0=dm(4,i2);
        f12 =f12 * f0;
        f12 = f12 - f13;
        f0 = f10 + f13,f3 = f10 - f13;
        f1 = f11 + f12,f2 = f11 - f12 ;
        f13 = f5 + f14, f10 = f5 - f14;
        f11 = f15 + f7, f12 = f15 - f7;
        f15 = f10;
        f7 = f11 + f13,f11 = f11 - f13,f14=dm(4,i2);
        f11 = f11 * f14;
        f5 = f10 + f12,f14=dm(5,i2) ;
        f5 = f5 * f14,f14=dm(6,i2);
        f10 = f12 * f14;
        f10 = f10 - f5,f14=dm(7,i2);
        f12 = f15 * f14;
        f12 = f12 + f5;
        f6 = f12 - f7;
        f5 = f11 -f6;
        f4 = f10 + f5;
        f15 = f0 + f7,r14=dm(8,i2);
        f15 = scalb f15 by r14;
        f15 = f1 + f6, dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        f15 = f2  + f5, dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        f15 = f3-f4,dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        f15 = f3 + f4,dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        f15 = f2 -f5, dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        f15 = f1 - f6,dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        f15 = f0 - f7,dm ( i1 , 1 ) = f15;
        f15 = scalb f15 by r14;
        dm ( i1 , 1 ) = f15;
iloop2: modify(i0, m0) ;
cas0:   ;
         l1 = 0 ;
         l2 = 0 ;

    !    push the registers back

         popr(1, r11) ;
         popr(2, r10) ;
         popr(3, r9) ;
         popr(4, r7) ;
         popr(5, r6) ;
         popr(6, r5) ;
         popr(7, r3) ;
         popr(8, r2) ;
         popr(9, r1) ;
         popi(10, i2) ;
         popi(11, i1) ;
         popi(12, i0) ;
         popm(13, m2) ;
         popm(14, m1) ;
         popm(15, m0) ;

         alter(15) ;

         leaf_exit ;

         .endseg ;


.segment /dm seg_dmda ;

.PRECISION = 32 ;

.VAR const[9] =
     0.707107,
      0.382683,
     0.541196,
     1.306562,
   1.414213,
    1.847759 ,
    1.082392 ,
     -2.613125,
     -3 ;

         .endseg ;
```

```
#include "asm_sprt.h"

.segment /pm seg_pmco ;

.global _getdat;

_getdat:

        entry ;
        dm(i7,-10)=r2;
        dm(-1,i6)=i13;

!       saving the registers:

      dm(-9,i6)=r1;
        i0 = r4;
        r2=r8;
      r4=ashift r2 by 7;
        r2=r12;
      r4=r4+r2;

        r2=r4;
        f5=float r2;
        r2 = ashift r4 by -2;
         r9=r2;
```

```
          f12 = float r2;
          r8=-2;
          f2=scalb f5 by r8;
          f8=f2;
        f2=f8-f12;
          f6=f2;
        f2=pass f2;

          if ne jump cas2 (DB);nop;nop;

          r2=i0;
          r4=r9;
        r8=r4;
        r1=r2+r8;
        i4=r1;
        r2=dm(i4,m5);
        r4=-16777216;
        r2=r2 and r4;
        r4=lshift r2 by -24;
           r0=r4;
          jump cas1 (DB);nop;nop;
cas2:

          f2=f6;
        f4= 0x3e800000;
        comp(f2,f4);
          if ne jump cas3 (DB);nop;nop;
          r2=i0;
          r4=r9;
        r8=r4;
        r1=r2+r8;
        i4=r1;
        r2=dm(i4,m5);
        r4=16711680;
        r2=r2 and r4;
        r4=lshift r2 by -16;
          r0 = r4;

          jump cas1 (DB);nop;nop;
cas3:

          f2=f6;
        f4= 0x3f000000;
        comp(f2,f4);
          if ne jump cas4 (DB);nop;nop;
          r2=i0;
          r4=r9;
        r8=r4;
        r1=r2+r8;
        i4=r1;
        r2=dm(i4,m5);
        r4=65280;
        r2=r2 and r4;
        r4=lshift r2 by -8;
        r0 = r4;

        jump cas1 (DB);nop;nop;

cas4:

        f2=f6;
      f4= 0x3f400000;
      comp(f2,f4);

        if ne jump cas5 (DB);nop;nop;
        r2=i0;
        r4=r9;
      r8=r4;
      r1=r2+r8;
      i4=r1;
      r2=dm(i4,m5);
      r4=255;
      r2=r2 and r4;
        r0 = r2;
        jump cas1 (DB);nop;nop;
cas5:
cas1:
  !     push the registers back
        r1=dm(-9,i6);
        i13=dm(m7,i6);
         exit ;

           .endseg ;

 #include "asm_sprt.h"

 .segment /pm seg_pmco;

 .global _putdat;

 _putdat:

         entry ;
         dm(i7,-11)=r2;
         dm(-1,i6)=i13;

 ! save the registers:

         dm(-9,i6)=r1;
         r2=i0;
         dm(-10,i6)=r2;

        i1 = r4;
        dm(-2,i6)=r4;
        dm(-3,i6)=r8;
        dm(-4,i6)=r12;
        r2=r8;
        r4=ashift r2 by 7;
        r2=r12;
        r4=r4+r2;
        r2=r4;
        f5=float r2;
        r2 = ashift r4 by -2;
        r9=r2;
        f12 = float r2;
        r8=-2;
        f2=scalb f5 by r8;
        f8=f2;
        f2=f8-f12;
        f6=f2;
        f2=pass f2;
        if ne jump cas0 (DB);nop;nop;
        r2=i1;
        r4=r9;
        r8=r4;
        r1=r2+r8;
        i0=r1;
        r2=dm(i0,m5);
        r4=16777215;
        r2=r2 and r4;
        r4=dm(1,i6);
        r8=lshift r4 by 24;
        r2=r2+r8;
        dm(i0,m5)=r2;
cas0:

        f4= 0x3e800000;
        comp(f2,f4);
        if ne jump cas1 (DB);nop;nop;
        r2=i1;
        r4=r9;
        r8=r4;
        r1=r2+r8;
        i0=r1;
        r2=dm(i0,m5);
        r4=-16711681;
        r2=r2 and r4;
        r4=dm(1,i6);
        r8=lshift r4 by 16;
        r2=r2+r8;
        dm(i0,m5)=r2;
 cas1:
        f4= 0x3f000000;
        comp(f2,f4);
        if ne jump cas2 (DB);nop;nop;
        r2=i1;
        r4=r9;
        r8=r4;
        r1=r2+r8;
        i0=r1;
        r2=dm(i0,m5);
        r4=-65281;
        r2=r2 and r4;
        r4=dm(1,i6);
        r8=lshift r4 by 8;
        r2=r2+r8;
        dm(i0,m5)=r2;
cas2:

        f4= 0x3f400000;
        comp(f2,f4);
        if ne jump cas3 (DB);nop;nop;
        r2=i1;
        r4=r9;
        r8=r4;
        r1=r2+r8;
        i0=r1;
        r2=dm(i0,m5);
        r4=-256;
        r2=r2 and r4;
        r4=dm(1,i6);
        r2=r2+r4;
        dm(i0,m5)=r2;
 cas3:



        r1=dm(-9,i6);
        i0=dm(-10,i6);
        i13=dm(m7,i6);

 exit;
.endseg;
#include "asm_sprt.h"


.segment /pm seg_pmco ;

.global     _getdaf;

_getdaf:

        entry ;
      dm(i7,-10)=r2;
      dm(-1,i6)=i13;

!       saving the registers:

      dm(-9,i6)=r1;
        i0 = r4;
      r2=r8;
        r4=ashift r2 by 7;
      r2=r12;
      r4=r4+r2;
        r2=r4;
      f5=float r2;

        r2 = ashift r4 by -1;
        r9=r2;
        f12 = float r2;
        r8=-1;
      f2=scalb f5 by r8;
        f8=f2;
      f2=f8-f12;
        f6=f2;
      f2=pass f2;
        if ne jump cas1 (DB);nop;nop;
        r2=i0;
      r4=r9;
      r8=r4;
      r1=r2+r8;
      i4=r1;
      r2=dm(i4,m5);
      r4=-65536;
      r2=r2 and r4;
      r4=lshift r2 by -16;
        r0=r4;

        jump cas3 (DB);nop;nop;
cas1:

        f2=f6;
      f4= 0x3f000000;
      comp(f2,f4);

        if ne jump cas3 (DB);nop;nop;

      r2=i0;
      r4=r9;
      r8=r4;
      r1=r2+r8;
      i4=r1;
      r2=dm(i4,m5);
      r4=65535;
      r0=r2 and r4;

        jump cas3 (DB);nop;nop;

cas3:
      r1=dm(-9,i6);
      i13=dm(m7,i6);

         exit ;

            .endseg ;
```

```
#include "asm_sprt.h"

.segment /pm seg_pmco;

.global     _putdaf;

_putdaf:

        entry;
      dm(i7,-11)=r2;
      dm(-1,i6)=i13;

!       saving the registers:

      dm(-9,i6)=r1;
      r2=i0;
      dm(-10,i6)=r2;
        i1=r4;
        r2=r8;
      r4=ashift r2 by 7;
      r2=r12;
      r4=r4+r2;
      r2=r4;
      f5=float r2;
        r2=ashift r4 by -1;
        r9=r2;
      f12 = float r2;
         r8=-1;
      f2=scalb f5 by r8;
        f8=f2;
      f2=f8-f12;
      f6=f2;
      f2=pass f2;

        if ne jump cas1 (DB);nop;nop;

        r2=i1;
      r4=r9;
      r8=r4;
      r1=r2+r8;
        i0=r1;
        r2=dm(i0,m5);
      r4=65535;
      r2=r2 and r4;
      r4=dm(1,i6);
      r8=lshift r4 by 16;
      r2=r2+r8;
        dm(i0,m5)=r2;
cas1:

      f2=f6;
      f4= 0x3f000000;
      comp(f2,f4);

        if ne jump cas3 (DB);nop;nop;

      r2=i1;
      r4=r9;
      r8=r4;
      r1=r2+r8;
        i0=r1;
        r2=dm(i0,m5);
      r4=-65536;
      r2=r2 and r4;
        r4=dm(1,i6);
        r2=r2+r4;
        dm(i0,m5)=r2;

cas3:
 ! push the registers back
      r1=dm(-9,i6);
      i0=dm(-10,i6);
      i13=dm(m7,i6);

exit;
```

```
.endseg;
```

```
#include "asm_sprt.h"

 .segment /pm seg_pmco ;

.global _dctproc;

_dctproc:

         entry;

      dm(i7,-9)=r2;
      dm(-1,i6)=i13;
```

```
!       saving the registers:

      dm(-7,i6)=r1;
      r2=i0;
      dm(-8,i6)=r2;
      dm(-2,i6)=r4;
      dm(-3,i6)=r8;
      dm(-4,i6)=r12;
      r2=0;
      dm(-5,i6)=r2;
cas0:
      r2=dm(-5,i6);
      r4=7;
      comp(r2,r4);

      if gt jump cas1 (DB);nop;nop;

      r2=0;
      dm(-6,i6)=r2;
cas3:
      r2=dm(-6,i6);
      r4=7;
      comp(r2,r4);

      if gt jump cas4 (DB);nop;nop;


      r2=dm(-5,i6);
      r4=ashift r2 by 3;
      r2=dm(-2,i6);
      r4=r4;
      r2=r2+r4;
      r4=dm(-6,i6);
      r8=r4;
      r1=r2+r8;
      i4=r1;
      f8=dm(i4,m5);
      r4=0xff800000;
      r2=logb f8;
      r2=-r2;
      r4=ashift r4 by r2;
      r4=r8 and r4;
      r4=fix f4;
      r8=abs r4;
      f2=float r8;
      r4=dm(-5,i6);
      r8=ashift r4 by 3;
      r4=dm(-3,i6);
      r8=r8;
      r4=r4+r8;
      r8=dm(-6,i6);
      r12=r8;
      r1=r4+r12;
      i4=r1;
```

```
f12=dm(i4,m5);
r8=0xff800000;
r4=logb f12;
r4=-r4;
r8=ashift r8 by r4;
r8=r12 and r8;
r8=fix f8;
r4=abs r8;
f8=float r4;
r4=dm(-5,i6);
r12=ashift r4 by 3;
r4=dm(-4,i6);
r12=r12;
r4=r4+r12;
r12=dm(-6,i6);
m4=r12;
i4=r4;
modify(i4,m4);
r4=dm(i4,m5);
r12=3;
r4=r4*r12 (ssi);
f12=float r4;
r4=-1;
f12=scalb f12 by r4;
f4=f8+f12;
comp(f2,f4);

if le jump cas6 (DB);nop;nop;


r2=dm(-5,i6);
r4=ashift r2 by 3;
r2=dm(-2,i6);
r4=r4;
r2=r2+r4;
r4=dm(-6,i6);
r8=r4;
r1=r2+r8;
i4=r1;
f2=dm(i4,m5);
f2=pass f2;

if le jump cas7 (DB);nop;nop;


r2=dm(-5,i6);
r4=ashift r2 by 3;
r2=dm(-2,i6);
r4=r4;
r2=r2+r4;
r4=dm(-6,i6);
r8=r4;
r1=r2+r8;
i4=r1;
```

```
        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-3,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r8=r4;
        r1=r2+r8;
        i0=r1;
        f8=dm(i0,m5);
        r4=0xff800000;
        r2=logb f8;
        r2=-r2;
        r4=ashift r4 by r2;
        r4=r8 and r4;
        r4=fix f4;
        r2=abs r4;
        f8=float r2;
        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-4,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r12=r4;
        r1=r2+r12;
        i0=r1;
        r2=dm(i0,m5);
        r4=3;
        r2=r2*r4 (ssi);
        f4=float r2;
        r2=-1;
        f12=scalb f4 by r2;
        f2=f8+f12;
        dm(i4,m5)=f2;

        jump cas8 (DB);nop;nop;

cas7:

        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-2,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r8=r4;
        r1=r2+r8;
        i4=r1;
        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-3,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r8=r4;
        r1=r2+r8;
        i0=r1;
        f8=dm(i0,m5);
        r4=0xff800000;
        r2=logb f8;
        r2=-r2;
        r4=ashift r4 by r2;
        r4=r8 and r4;
        r4=fix f4;
        r2=abs r4;
        f8=float r2;
        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-4,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r12=r4;
        r1=r2+r12;
        i0=r1;
        r2=dm(i0,m5);
        r4=3;
        r2=r2*r4 (ssi);
        f4=float r2;
        r2=-1;
        f12=scalb f4 by r2;
        f2=f8+f12;
        f4= -f2;
        dm(i4,m5)=f4;
cas8:

        jump cas9 (DB);nop;nop;

cas6:

        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-2,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r8=r4;
        r1=r2+r8;
        i4=r1;
        f8=dm(i4,m5);
        r4=0xff800000;
        r2=logb f8;
        r2=-r2;
        r4=ashift r4 by r2;
        r4=r8 and r4;
        r4=fix f4;
        r8=abs r4;
```

```
f2=float r8;
r4=dm(-5,i6);
r8=ashift r4 by 3;
r4=dm(-3,i6);
r8=r8;
r4=r4+r8;
r8=dm(-6,i6);
r12=r8;
r1=r4+r12;
i4=r1;
f12=dm(i4,m5);
r8=0xff800000;
r4=logb f12;
r4=-r4;
r8=ashift r8 by r4;
r8=r12 and r8;
r8=fix f8;
r4=abs r8;
f8=float r4;
r4=dm(-5,i6);
r12=ashift r4 by 3;
r4=dm(-4,i6);
r12=r12;
r4=r4+r12;
r12=dm(-6,i6);
m4=r12;
i4=r4;
modify(i4,m4);
r4=dm(i4,m5);
r12=3;
r4=r4*r12 (ssi);
f12=float r4;
r4=-1;
f12=scalb f12 by r4;
f4=f8-f12;
comp(f2,f4);

if ge jump cas10 (DB);nop;nop;


r2=dm(-5,i6);
r4=ashift r2 by 3;
r2=dm(-2,i6);
r4=r4;
r2=r2+r4;
r4=dm(-6,i6);
r8=r4;
r1=r2+r8;
i4=r1;
f2=dm(i4,m5);
f2=pass f2;
if le jump cas11 (DB);nop;nop;

r2=dm(-5,i6);
```

```
      r4=ashift r2 by 3;
      r2=dm(-2,i6);
      r4=r4;
      r2=r2+r4;
      r4=dm(-6,i6);
      r8=r4;
      r1=r2+r8;
      i4=r1;
      r2=dm(-5,i6);
      r4=ashift r2 by 3;
      r2=dm(-3,i6);
      r4=r4;
      r2=r2+r4;
      r4=dm(-6,i6);
      r8=r4;
      r1=r2+r8;
      i0=r1;
      f8=dm(i0,m5);
      r4=0xff800000;
      r2=logb f8;
      r2=-r2;
      r4=ashift r4 by r2;
      r4=r8 and r4;
      r4=fix f4;
      r2=abs r4;
      f8=float r2;
      r2=dm(-5,i6);
      r4=ashift r2 by 3;
      r2=dm(-4,i6);
      r4=r4;
      r2=r2+r4;
      r4=dm(-6,i6);
      r12=r4;
      r1=r2+r12;
      i0=r1;
      r2=dm(i0,m5);
      r4=3;
      r2=r2*r4 (ssi);
      f4=float r2;
      r2=-1;
      f12=scalb f4 by r2;
      f2=f8-f12;
      dm(i4,m5)=f2;
      jump cas12 (DB);nop;nop;
cas11:

      r2=dm(-5,i6);
      r4=ashift r2 by 3;
      r2=dm(-2,i6);
      r4=r4;
      r2=r2+r4;
      r4=dm(-6,i6);
      r8=r4;
      r1=r2+r8;
```

```
        i4=r1;
        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-3,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r8=r4;
        r1=r2+r8;
        i0=r1;
        f8=dm(i0,m5);
        r4=0xff800000;
        r2=logb f8;
        r2=-r2;
        r4=ashift r4 by r2;
        r4=r8 and r4;
        r4=fix f4;
        r2=abs r4;
        f8=float r2;
        r2=dm(-5,i6);
        r4=ashift r2 by 3;
        r2=dm(-4,i6);
        r4=r4;
        r2=r2+r4;
        r4=dm(-6,i6);
        r12=r4;
        r1=r2+r12;
        i0=r1;
        r2=dm(i0,m5);
        r4=3;
        r2=r2*r4 (ssi);
        f4=float r2;
        r2=-1;
        f12=scalb f4 by r2;
        f2=f8-f12;
        f4= -f2;
        dm(i4,m5)=f4;
cas12:
cas10:
cas9:
cas5:

        r2=dm(-6,i6);
        r4=r2+1;
        dm(-6,i6)=r4;

        jump cas3 (DB);nop;nop;

cas4:


cas2:
        r2=dm(-5,i6);
        r4=r2+1;
```

```
      dm(-5,i6)=r4;

      jump cas0 (DB);nop;nop;

cas1:


!      push the registers

      r1=dm(-7,i6);
      i0=dm(-8,i6);
      i13=dm(m7,i6);



 exit;


.endseg;
```